BEST AVAILABLE IMAGES

Defective images within this document are accurate representations of the original documents submitted by the applicant.

Defects in the images may include (but are not limited to):

- BLACK BORDERS
- TEXT CUT OFF AT TOP, BOTTOM OR SIDES
- FADED TEXT
- ILLEGIBLE TEXT
- SKEWED/SLANTED IMAGES
- COLORED PHOTOS
- BLACK OR VERY BLACK AND WHITE DARK PHOTOS
- GRAY SCALE DOCUMENTS

IMAGES ARE BEST AVAILABLE COPY.

As rescanning documents *will not* correct images, please do not report the images to the Image Problem Mailbox.

(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
12 September 2002 (12.09.2002)

PCT

(10) International Publication Number
WO 02/071211 A2

(51) International Patent Classification[7]: G06F 9/40

(21) International Application Number: PCT/US01/44031

(22) International Filing Date:
20 November 2001 (20.11.2001)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:

60/252,170	20 November 2000 (20.11.2000)	US
60/256,550	18 December 2000 (18.12.2000)	US
60/270,696	22 February 2001 (22.02.2001)	US
60/276,375	16 March 2001 (16.03.2001)	US
60/290,520	11 May 2001 (11.05.2001)	US
60/323,022	14 September 2001 (14.09.2001)	US
PCT/US01/43829	20 November 2001 (20.11.2001)	US
PCT/US01/43444	20 November 2001 (20.11.2001)	US
PCT/US01/44035	20 November 2001 (20.11.2001)	US
PCT/US01/43957	20 November 2001 (20.11.2001)	US

(71) Applicant: **ZUCOTTO WIRELESS, INC.** [US/US]; 4225 Executive Square, Suite 400, La Jolla, CA 92037 (US).

(72) Inventors; and

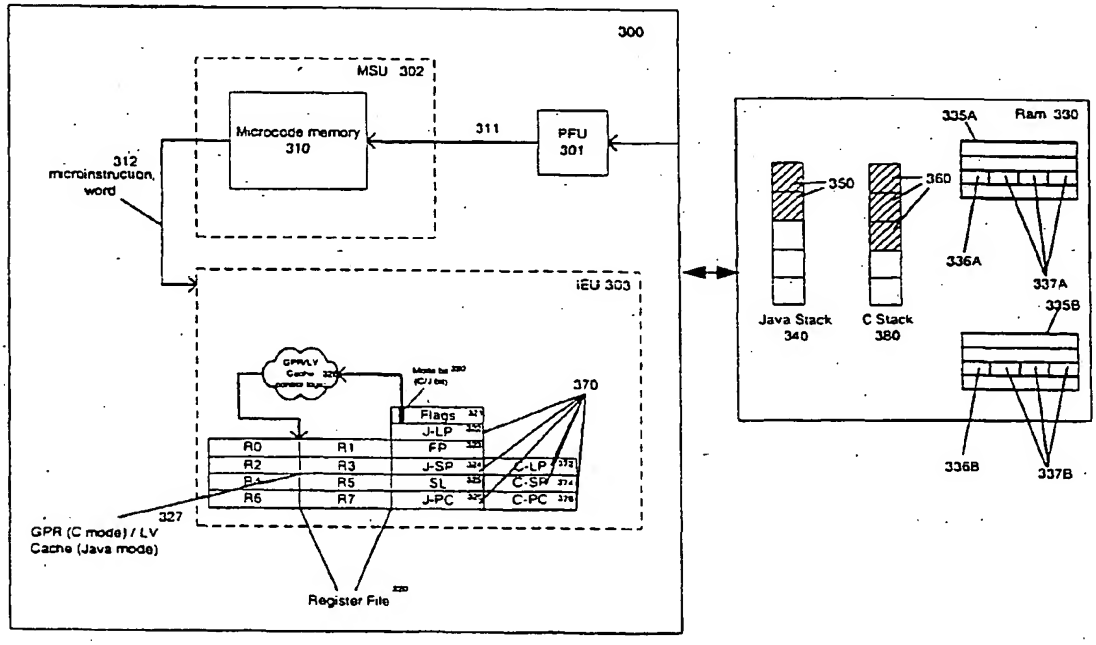
(75) Inventors/Applicants *(for US only)*: **MAJID, Michael** [CA/CA]; 106 Halley Str., Nepean, Ontario K2J3R9 (CA). **SAHRAOUI, Zohair** [CA/CA]; 6243 Castille Court, Gloucester, Ontario K1C1X4 (CA). **COUTURE, Mark** [CA/CA]; 13 Mystic Pvt., Ottawa, Ontario K1V1K5 (CA). **ROUFFER, Christian** [CA/CA]; 133 Hamilton Ave., Ottawa, Ontario K1Y1C1 (CA).

(74) Agent: **WARDAS, Mark**; 4225 Executive Square, Suite 400, La Jolla, CA 92037 (US).

(81) Designated States *(national)*: AE, AG, AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, BZ, CA, CH, CN, CO, CR, CU, CZ, DE, DK, DM, DZ, EC, EE, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MA, MD, MG, MK, MN, MW,

[Continued on next page]

(54) Title: DATA PROCESSOR HAVING MULTIPLE OPERATING MODES



(57) Abstract: Registers are provided for fast mode switching between a first mode and a second mode and to facilitate operations in one mode that need to manipulate a context of another mode.

WO 02/071211 A2

MX, MZ, NO, NZ, PH, PL, PT, RO, RU, SD, SE, SG, SI, SK, SL, TJ, TM, TR, TT, TZ, UA, UG, US, UZ, VN, YU, ZA, ZW.

(84) **Designated States** *(regional)*: ARIPO patent (GH, GM, KE, LS, MW, MZ, SD, SL, SZ, TZ, UG, ZM, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE, TR), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, ML, MR, NE, SN, TD, TG).

Declaration under Rule 4.17:

— *of inventorship (Rule 4.17(iv)) for US only*

Published:

— *without international search report and to be republished upon receipt of that report*

For two-letter codes and other abbreviations, refer to the "Guidance Notes on Codes and Abbreviations" appearing at the beginning of each regular issue of the PCT Gazette.

Data Processor Having Multiple Operating Modes

Related Applications

This application claims priority and is related to commonly assigned
PCT Application S.N. Docket number 1065PCT, filed on this day concurrently;
PCT Application S.N. Docket number 1071PCT, filed on this day concurrently;
US Provisional Application S.N. 60/276,375 Docket Number 1065.1;
US Provisional Application S.N. 60/252,170 Docket Number 1065;
US Provisional Application S.N. 60/323,022 Docket Number 1123;
US Provisional Application S.N. 60/290,520 Docket Number 1024.1;
US Provisional Application S.N. 60/270,696 Docket Number 1024; and
US Provisional Application S.N. 60/256,550 Docket Number 1089; which are all incorporated herein by reference.

Field of the Invention

The present invention relates to use of processors more than one mode, and in particular to data processors having multiple operating modes, each mode utilizing different instruction sets.

Background

Java[TM] technology-based native processors are enabling embedded devices such as cellular phones, pagers, personal digital assistants, industrial controllers, set-top boxes, and Internet appliances with new functionality.

To ensure compatibility and proper execution of Java bytecode, Java technology-based native processors may implement a standardized instruction set defined by the Java Virtual Machine specification (http://java.sun.com/docs/books/vmspec/html/VMSpecTOC.doc.html). The JVM instruction set in combination with its run-time data structures define a "sandbox" that provides a secure run-time environment for which Java is well known. For example, the main memory of a system is essentially presented to Java applications as a black box – objects may be created and destroyed, however a Java developer need not know, nor may control, specific memory addresses that contain object data and references. Accordingly, the JVM instruction set does not need to include opcodes and addressing modes for addressing specific memory locations. Instead, the JVM implements a stack-based machine, providing one or more run-time stacks (additional stacks may be provided for additional threads of execution) for implicit storage of various data and handles (or references) to data allocated in a predefined heap in memory.

On the other hand, certain programs, such as system-level code for interrupt service routines (ISRs) and other OS-like functionality, may need to access specific memory addresses, including memory-mapped peripherals. Thus, Java native processors may need to implement the standard JVM instruction set as well as a set of extended instructions to provide full functionality.

Accordingly, some Java native processors define two modes of operation: a Java mode, wherein the processor executes Java bytecode using only the JVM instruction set and optimized variants of JVM instructions, and an extended mode, wherein the processor executes applications using the extended instruction set. Interrupt service routines, exceptions, and traps are generally handled in extended mode.

The extended mode may comprise a set of instructions and run-time structures (i.e. the JVM + extended instructions or just the extended instructions) such that one or more high level-languages may be compiled to the instruction set of the Java native processor or non-Java processor core for execution thereon.

Additionally, although Java native processors may implement most opcodes in hardware (i.e. via microcode or hardwired control), it is often advantageous to implement certain opcodes in software – so-called "trapped opcodes", or "emulated opcodes". Trapped opcodes may be implemented by a trap handler stored in system memory, and may be executed in extended mode.

EXAMPLE OF PROCESSOR CORE

Figure 1 illustrates a conventional multi-mode processor core of a Java native processor that may execute applications in either a Java mode or a C language mode. Multi-mode core 100 comprises a pre-fetch unit (PFU) 101, microsequencer unit (MSU) 102 having a microcode memory 110, and an instruction execution unit (IEU) 103 having a register file 120. Multi-mode core 100 is coupled to system memory 130, which contains one or more programs 135.

System memory 130 is provided to store programs 135 (i.e. instructions) and data as well as a run-time stack 140 for use in both Java and C modes. Program 135 comprises a plurality of instructions, with each instruction comprising an opcode 136, which may or may not be associated with one or more operands 137. Run-time stack 140 is generally utilized for

maintaining a call chain of method frames (aka activation records), for passing parameters to methods (or functions), returning results to callers, and for holding temporary results, etc. Run-time stacks are well known to those skilled in the art.

Register file 120 may comprise a plurality of specific purpose registers, including a status flag (Flags) register 121 for indicating various conditions during execution; local variable pointer (LP) 122, for pointing the local variables of the current execution context; frame pointer (FP) 123, for pointing to the return execution context; stack pointer (SP) 124, for pointing to top of the stack (TOS)140 in system memory 130; stack limit pointer (SL) 125, for pointing to the end of the current stack 140; and program counter (PC) 126, for pointing to the next instruction to be fetched. Register file 120 may also include a set of general-purpose registers (GPRs) 127. Any of the general-purpose registers 127 may serve a specific purpose for certain instructions and serve as general purpose registers for other instructions, primarily for use in extended mode.

Pre-fetch unit 101 fetches instructions from programs 135 in system memory 130 according to a value contained in program counter (PC) 126 register, or according to an instruction pre-fetch pointer stored in another register (not shown).

Microcode memory 110 receives a microprogram address 111 from PFU 101, corresponding to an opcode to be executed, to obtain one or more microinstructions comprising the microprogram that implements the opcode. The microinstructions are dispatched to IEU 103 to control the datapath and effect execution of the opcode.

SINGLE RUN-TIME STACK WITH JAVA AND C FRAMES

Figure 2A illustrates details of a run time stack 140 of a conventional Java native processor. Run-time stack 140 is illustrated as a single contiguous region of memory in system memory 130, however the stack may also comprise a series of linked stack chunks using a technique known as "stack chunking". Linked stack chunks may either be contiguous or discontiguous chunks. In a multi-threaded system, a separate run-time stack 140 may be allocated for each thread of execution.

Stack 140 includes multiple "stack frames", each comprising the execution context and related data of a single method invocation or function call. In the illustrated example (representing a prior art multi-mode system), a C frame 160 and a Java frame 150 are located

on stack 140. Java frame 150 is the "active frame", indicated by the various pointers (122, 123, and 124) pointing into its memory space, as it is associated with the current execution context in core 100.

The exact structure of each mode's frame is, at least to some extent, implementation as well as mode dependant. Generally, the method frame structures are defined by the nature of the system architecture, the execution mode, and the data required to reinstate the execution context on processor core 100. Thus, the structure of Java frame 150 may differ from C frame 140.

In the example illustrated in Figure 2A, Java frame 150 comprises three method frame segments: local variable segment 151, "return execution context" segment 152, and local stack operand segment 153.

Local variable segment 151 comprises local variables required for the execution of the method. Local variable segment 151 may also include any parameters passed to the method by its caller. If parameters are passed to a method from its caller, the caller will first push the parameters onto its local operand stack segment 153 in stack memory 140. When the caller invokes a Java method, an invoke instruction is executed. The invoke instruction may include construction of a new Java frame 150 for the invoked method on stack memory 140, such that new Java frame 150 overlaps with any parameters in the local stack operand segment 153 in the caller's method frame. Accordingly, the invoked Java method may access the parameters as local variables.

The return execution context segment 152 typically contains the information required to restore the execution context of the caller when the method returns program control thereto (i.e. upon the execution of a return instruction). Typically, execution context segment 152 will include a plurality of elements, including a return program counter (PC), return frame pointer (FP), return local variable pointer, constant pool pointer, current method index, and current class index.

The return execution context segment 152 is constructed by pushing each of its elements onto stack memory 140. As indicated in Figure 2A, Java frame 150 is indicated as active frame by the various pointers (122, 123, and 124), such that Java frame 150 is associated with the current execution context.

Local stack operand segment 153 is defined in Java frame 150 to accommodate temporary results and to pass parameters to methods and receive return values from methods invoked by the method associated with the current context. In Java environments, these temporary operands are known as local operands, stack operands, or local stack operands. Local stack operands may comprise constants, for example, local variables, fields, references, results, etc. As processor core 100 executes instructions of a method, local stack operand segment 153 may grow and shrink as certain instructions, such as the JVM opcodes i_const0, aload, astore, and the like, push and pop local stack operands onto stack memory 140. For example, the JVM iadd instruction pops the top two elements off local operand stack segment 153, adds the two elements in an ALU in processor 100 and generates a sum, and pushes the sum onto the local operand segment 153 (i.e.: the top of stack). The sum may be subsequently stored in a local variable, serve as an operand to a subsequent instruction, or be returned to the caller as a return value. Stack pointer 124 indicates the location of the top element (TOS) of the local stack operand. Memory locations beyond stack pointer 124 contain invalid data.

In the illustrated example of Figure 2A, C frame 160 differs in structure from Java frame 150. C frame 150 may comprise parameters 161, return PC 162, return local variable pointer 163, and local operand stack 164. The C frame of the present example generally corresponds to that of the picoJava processor referenced in picoJava-II™ Programmer's Reference Manual, March 1999, available from Sun Microsystems, Palo Alto, CA. When a C frame is the current frame, the local variable pointer 122 points to the bottom of the frame. Where at least one parameter is provided, the bottom comprises a parameter, otherwise the bottom comprises return program counter 162. Stack pointer 124 points to the TOS in the local stack operand segment 164. The stacks in the Figures are oriented such that the stack bottom is located towards the top of the page, while the top of the stack is located towards the bottom of the page. Also, where multiple frames are illustrated on a stack, the TOS comprises the address of valid data located closest to the top of the stack (i.e. closest to the bottom of the page).

As a call chain develops at run-time through a series of C-function calls, Java method invocations, traps, and C and Java mode frames are pushed onto stack memory 140. Some

systems, particularly embedded systems, utilize stack chunking to provide dynamic allocation of stack memory whenever a call chain exhausts the memory allocated to a stack chunk. In this way, memory may be conservatively allocated to threads on an as-needed basis. When a given thread requires additional memory, another stack chunk is allocated and linked to the previous stack chunk. The picoJava processor utilizes this method of dynamic stack allocation.

When a new stack chunk is allocated, a stack limit pointer 125 is updated to point to the last available memory address in the allocated stack chunk. Every time data is pushed onto the stack, the stack limit must be checked to determine whether the stack chunk can accommodate the data. If insufficient allocated memory remains in the current stack chunk, program execution is interrupted and a stack chunk trap handler executes. The stack chunk trap handler attempts to allocate memory for a new stack chunk and performs administrative tasks to ensure proper linkage between the new stack chunk and previous stack chunk. Disadvantageously, the stack chunking trap handler itself requires a C frame to be pushed onto stack memory 140 so that program execution may return correctly to the context that caused the stack-chunking trap.

STACK CHUNKING

Figure 2B illustrates a conventional stack chunk 205A and current stack chunk 205B, comprising portions of stack memory 140, where a stack chunking trap handler is called. Illustrated in Figure 2B is the invocation of a Java method, the Java method having Java frame 220. In Figure 2B stack chunk 205B cannot accommodate Java frame 220, as is determined by comparing the Java frame stack requirements with the difference between stack pointer 124 and stack limit 125. In Figure 2B, execution is trapped, jumping to the stack chunking trap handler. The stack chunking trap handler begins to build a C frame 210 on stack chunk 205B one element at a time. However, there may be insufficient allocated memory remaining in stack chunk 205B to accommodate trap frame 210. To avoid this problem, prior art multi-mode processors need to be designed to ensure that sufficient allocated memory remains on stack chunk 205B to accommodate the stack chunk trap frame. Because this solution reserves enough stack memory for a stack chunk trap frame, stack-

chunking traps may occur prematurely, the stack chunking operation may result in inefficient use of stack memory.

Although stack chunking may provide a benefit of conservative memory management, which is an important consideration in embedded systems such as cellular phones and other mobile or stationary devices, C mode operations incur substantial overhead when operating on a stack chunk. Generally, unlike standard Java compilers, C language compilers do not generally generate information from which one may determine the maximum stack requirements of a function at compile time. Consequently, every stack operation may be required to check the stack limit to avoid stack overflow, thereby reducing performance with frequent stack limit checking. Because ISRs are typically implemented in C mode in Java native processors, time-critical ISRs may be particularly affected by the overhead of stack limit checking when stack chunking is utilized. Furthermore, the possibility of a stack chunk boundary being crossed at any given time and the overhead of transition to operation in another chunk (either returning to a previous chunk, or continuing onto a new or next chunk) adds to the worst-case interrupt latency of ISRs and other C mode operations.

TRAPPED OPCODES

Figure 2C illustrates a prior art memory mapping of a stack 140 (Fig. 2b) during the execution of a trap handler for a trapped opcode (i.e. an opcode implemented by trapping to a software trap handler). In the illustrated example, a call chain has developed in stack memory, including C frame 160, Java frame 150, and a C frame 260 corresponding to a trap handler used to emulate the trapped opcode. To facilitate trap handler code development, the trap handler code may be implemented using a collection of C language functions. Consequently, the trap handler executes, in C mode, a series of C mode function calls, resulting in C frames 265, 270 and 275 in stack memory. C frame 275 is the active frame, providing a return program counter value (PC) 276, return local variable pointer 277, and a local operand stack 278. Stack pointer 124 points into local operand stack 278 and local variable pointer 122 points to return local variable pointer 277.

Illustrated in Figure 2C is the trapped instruction 235 that caused the initial trap to occur. Trapped instruction 235 is a Java instruction (emulated by a trap handler) that may be

stored in a method structure (for example, a JVM data structure) in system memory 130. As shown, trapped instruction 235 comprises opcode 236 and multiple operands 237.

To emulate the instruction, the trap handler may have to perform any of the following operations:

- access the opcode (quickening);
- access the operands (all cases where operands are provided & quickening);
- push or pop data on the Java frame associated with the context where the trap occurred (invokes, returns, getfield);
- access the stack pointer and stack limits (invokes);
- read or write a local variable from the Java frame associated with the context where the trap occurred.

Disadvantageously, previously existing methods for handling the above operations in a multi-mode system are complicated, inefficient, and may result in non-deterministic behavior.

For example, for all stack operations, such as pushes, pops, and other stack manipulations, the stack pointer associated with the trapped execution context (i.e. the context of the method in which the trapped instruction is located) needs to be accessed. However, as the stack pointer register of that context may be saved in the trap frame (C frame), access to the Java frame stack pointer may require at least one level of indirection, or O(n) behavior (known as "big-O notation" in the arts) (in the case of a C-mode call chain required to implement the trapped opcode), where n is the number of C frames that need to be searched to find the saved Java stack pointer.

Furthermore, to push a value onto Java frame 150, the return PC 261 and subsequent elements may be overwritten by the pushed data. To overcome this problem, trap handlers in multi-mode processors would either need to somehow know how many stack entries would be modified in the emulation of the opcode to "pad" the stack with empty elements between the Java frame 150 and the trap frame 260. Another alternative might be to shift all stack elements down the stack every time data is pushed. This solution may reduce performance of the system and increase power dissipation due to toggling memory lines and busses that affect the copy steps. While the former is almost always a concern, the latter is of particular concern

in devices that comprise battery powered embedded systems, for example, such as wireless mobile devices.

Again, related to stack operations, certain trapped opcodes may require access to the stack pointer to compare against the stack limit pointer for stack chunking, such as the Java invokeVirtual opcode (instruction used to invoke certain methods).

Access to the opcode that caused the trap may be required in order to determine the location of the trap in a trap table (as opcode emulating trap handlers may be indexed using the opcode). Other trapped opcodes may be self-modifying and replace their instance at run-time with a so-called "quick" version of the opcode. Accordingly, the value of the program counter 126 prior to the call to the trap handler would be required. This is available via return PC 261, however, as the case with the stack pointer, at least one level of indirection would be incurred as well as the possibility of an O(n) search through a C frame call chain caused by function calls within the trap handler code.

It should be noted that in certain implementations, the Java stack pointer may be provided as a function of the trap handler's local variable pointer. However, even in these cases, the O(n) search behavior would still apply. Coding the trap handlers without the use of function calls, could mitigate this, however such an approach would be difficult to develop and maintain. Such a strategy would prevent efficient reuse of code within a program and would grow the size of the code. Furthermore, trapped opcodes that require access to the local variables in the local variable segment of Java frame 150 would need to seek out the saved local variable pointer. The same shortcomings would be found in local variable operations in trapped opcodes.

PRIOR ART MODE (CONTEXT) REGISTERS

While not related to multi-mode systems certain prior art processors provide multiple sets of registers, wherein each set is associated with a separate execution context. Some architectures such as the UltraSparc from Sun Microsystems, Palo Alto, CA provide multiple sets of context-specific registers. While such sets of registers provide fast context switching, they do not facilitate access to a first mode from a second mode and vice-versa.

What is needed therefore are improvements and solutions to one or more of the above-identified disadvantages.

Summary of the Invention

In one embodiment, registers are provided for fast mode switching between a first mode and a second mode and to facilitate operations in one mode that need to manipulate a context of another mode. For example, in a Java native processor, implementing the instructions of the JVM instruction set and a set of extended instructions, a first mode may comprise a Java mode, and a second mode may comprise a C mode. In one embodiment, a Java native processor may implement certain opcodes of a JVM instruction set in hardware in a Java mode, while a second set of opcodes of the JVM instruction set are trapped, or trappable, and may handled by a trap-handler. The trap-handler, requiring functionality beyond what is provided by the instruction available in the JVM instruction set, may comprise a sequence of one or more extended instructions (with respect to the JVM instruction set) and the aforementioned hardware implemented JVM instructions. Trap handlers may either be developed in native assembler or in any suitable programming language for which a compiler is provided to compile the language to the native instruction set of the target system. In one embodiment, trap handlers are developed in the C programming language, thus extended mode operations are known as "C mode" operations. Hereafter, the latter term will be used, describing embodiments of the present invention in the context of a Java native processor having both a Java mode and a C mode, but other languages are understood to be possible for other modes. C mode may also be referred to as extended mode.

Embodiments of the invention provide efficient execution of program code comprising more than one instruction set. In one embodiment, the present invention provides efficient operation in both Java modes and C modes of a Java native processor, Java hardware accelerator or other hardware Java solutions.

In one embodiment, a dual-mode system for executing instructions may comprise: a first stack and a second stack; a circuit adapted to execute instructions in a first mode using the first stack; and a circuit adapted to execute instructions in a second mode using the first stack and the second stack. The first and second stack may be adapted to store information. The instructions executed in the first mode may operate on information stored on the first stack, and he instructions executed in the second mode may operate on information stored on the first stack and the second stack. The circuit adapted to execute instructions in a first mode,

and the circuit adapted to execute instructions in a second mode may comprise a data processor. The circuit adapted to execute instructions in a first mode and the circuit adapted to execute instructions in a second mode may comprise a data processor coupled to a hardware instruction accelerator. The hardware instruction translator may comprise an instruction accelerator adapted to utilize a JVM instruction set. The data processor may comprise a Java technology-based native processor. The system may further comprise one or more memories, the memories operatively coupled to the circuit adapted to execute instructions in a first mode and the circuit adapted to execute instructions in a second mode, the memories storing the first stack and the second stack. The first stack may comprise one or more stack chunks. The second stack may comprise a statically allocated stack. The instructions executed in the first mode may comprise software instructions executed in a platform independent mode, and the instructions executed in the second mode may comprise software instructions executed in a platform dependant mode. The instructions executed in the first mode may comprise software instructions executed in a Java mode, and the instructions executed in the second mode may comprise software instructions executed in a platform dependant mode. The platform dependant mode may provide Java instruction emulation for the software instructions executed in the platform dependant mode. The platform dependant mode may comprise a C mode, the instructions executed in the C mode comprising instructions compiled from one or more C language functions to a native instruction set of the system. The instructions executed in the first mode may comprise object-oriented software instructions executed in an object-oriented mode, and the instructions executed in the second mode may comprise software instructions executed in a platform dependant mode, the platform dependant mode providing object-oriented software instruction emulation. The one or more memories may define one or more memory locations for use in a first mode and storing information related to an execution context of the first mode, the one or more computer readable medium defining one or more memory locations for use in a second mode and storing information related to an execution context of the second mode, and the one or more memory locations for use in a first mode being available and addressable to software instructions operating in the second mode. The one or more memory locations may be used in a first mode and the one or more memory locations may be used in a second mode storing information comprising one or more values

selected from a program counter, a stack pointer, a local variable pointer, and a frame pointer. The one or more memory locations may be used in a first mode comprising one or more registers, and the one or more memory locations may be used in a second mode comprising one or more registers. The one or more memories may define a mode indicator for indicating and controlling whether the system executes instructions in the first mode or in the second mode. The system may further comprise a mode indicator adapted to control which of the circuit of the circuit adapted to execute instructions in a first mode and the circuit adapted to execute instructions in a second mode is active, the mode indicator having a first state indicating the circuit adapted to execute instructions in a first mode is active and a second state indicating the circuit adapted to execute instructions in a second mode is active. Access to the one or more memory locations for use in a first mode and access to the one or more memory locations for use in a second mode may be controlled by a status of the mode indicator. The system may be adapted to maintain a prior mode indicator history such that after completion of an operation in a current mode a prior mode can be restored. The prior mode indicator history may comprise a return mode indicator. The return mode indicator may be stored in an element in a return execution context on one of the first stack and the second stack, and the one of the first stack and the second stack may be selected by the prior mode indicator. The element storing the return mode indicator may comprise a return flags element. The element in a return execution context may include one or more bits for representing the return mode indicator. The operation may comprise the execution of a return instruction. The system may further comprise: a mode switch control adapted to set the mode indicator to the first state upon occurrence of any one of a first set of conditions and adapted to set the mode indicator to the second state upon occurrence of any one of a second set of conditions. The first set of conditions may comprise one or more of the group comprising: a software trap; a hardware trap; the servicing of an interrupt; while in Java mode, a native method call; and an explicit software controlled change of the state of the mode indicator to set the state to the second state. The second set of conditions may comprise one or more of the group comprising: the execution of a return instruction, the return instruction returning from a trap handler; the execution of a return instruction, the return instruction returning from an interrupt

handler; the execution of a return instruction, the return instruction returning from a native method, the native method previously called from an instruction executed in the first mode; and an explicit software controlled change of the state of the mode indicator to set the state to the second state. The first set of conditions may comprise a hardware trap, wherein the hardware trap comprises a trap to an instruction emulation trap, the instruction emulation trap emulating an instruction selected from the group comprising: an object-oriented instruction, a platform independent instruction, and a Java instruction.

In one embodiment, a system including a processor having multiple execution modes may comprise: a mode indicator adapted to indicate one of a plurality of execution modes; and a memory storing at least one set of values, each set of values having a respective member for each mode for each of the plurality of execution modes, wherein the mode indicator is further adapted to select the respective member for the one of the plurality of execution modes for each set of values. The system may comprise: a first set of registers specific to the first mode; and a second set of registers specific to the second mode, wherein the first set of registers is addressable when said mode indicator is in the first state, and the second set of registers is addressable when said mode indicator is in the second state, and wherein the first set of registers is addressable when said mode indicator is in the second state. The system may comprise multiplexing circuitry adapted to select either the first set of registers or the second set of registers as a function of the mode indicator. The system may comprise: a set of current registers coupled to the output of the multiplexing circuitry; and a set of alternate registers coupled to the output of the multiplexing circuitry, the multiplexing circuitry selecting the first set of registers as the content of the current registers when the mode indicator is set to indicate the first state, the multiplexing circuitry selecting the second set of registers as the content of the alternate registers when the mode indicator is set to indicate the first state, the multiplexing circuitry selecting the second set of registers as the content of the current registers when the mode indicator is set to indicate the second state, and, the multiplexing circuitry selecting the first set of registers as the content of the alternate registers when the mode indicator is set to indicate the second state. The system may comprise: at least one instruction which is used in both the first and second mode which when executed is adapted to access one or more of the registers in the set of current registers.

In one embodiment, a dual-mode system for executing instructions may comprise at least one set of registers, each set of registers being used as a cache in a first mode and as a general-purpose register in a second mode. The set of registers may be capable of being used as a cache in the first mode and as a general-purpose register in the second mode. The cache may comprise a local variable cache. The cache may comprise a Java local variable cache. The Java local variable cache may be a direct-mapped cache or an associative cache.

In one embodiment, an instruction decoder may be adapted to employ one or more mode bits supplied as the highest order bits of a decoder input in combination with fixed length opcodes to increase the number of possible opcode values.

In one embodiment, a multi-mode system for executing instructions may comprise: an instruction fetch unit for receiving instructions; a mode indicator adapted to indicate one of the plurality of modes, the mode indicator comprising one or more bits; an instruction decoder coupled to the instruction fetch unit and adapted to the decode the instructions as an n-bit opcode, the instruction decoder comprising an input adapted to receive the n-bit opcode and the mode indicator, the one or more bits of the mode indicator comprising one or more highest order bits of the decoder input; and a processor coupled to the instruction fetch unit and adapted to perform a first set of one or more operations when an opcode of an instruction is decoded in the first mode, and adapted to perform a second set of one or more operations when an opcode of an instruction is decoded in the second mode. The instruction decoder may be selected from a group comprising: a software decoder, a hard-wired logic circuit decoder, and a memory-based decoder. The instruction decoder may comprise a hard-wired logic circuit decoder, the hard-wired logic circuit decoder selected from the group comprising: a PLA circuit; a PAL circuit; and a circuit, the circuit comprising one or more gates. The instruction decoder may comprise a memory-based decoder, the memory-based decoder selected from the group comprising: a ROM circuit storing a look-up table, a RAM circuit storing a look-up table, and a Flash memory circuit storing a look-up table. The instruction decoder may be integral with the processor. The processor may comprise a Java technology-based native processor. The processor may comprise an object-oriented processor. The instruction decoder may comprise a Java technology-based accelerator circuit. The first set of one or more operations may comprise one opcode of a first instruction set, and the second set

of one or more operations may comprise one opcode of a second instruction set. The plurality of modes may comprise two modes, the mode indicator comprising one mode bit, the decoder decoding up to 2^{n+1} possible opcodes, wherein the first instruction set comprises one or more of the first 2^n opcodes of the 2^{n+1} possible opcodes, and wherein the second instruction set comprises one or more of the 2^n+1 to 2^{n+1} opcodes of the 2^{n+1} possible opcodes. In one embodiment n is equal to 8. The second mode may comprise a platform dependant mode and the second instruction set may comprise a platform dependant instruction set. The first mode may comprise a Java mode and the first instruction set may comprise an instruction set specified in the Java Virtual Machine specification. The first instruction set may comprise one or more quick instructions. The first instruction set may comprise one or more folded instructions. The platform dependant instruction set may comprise one or more opcodes for accessing specific memory locations. The platform dependant instruction set may comprise one or more opcodes for providing register-based operations. The platform dependant mode may comprise a C mode, wherein the instructions executed in the platform dependant mode comprise instructions compiled from the C programming language to one or more instructions of the platform dependant instruction set.

In one embodiment, a multi-mode system for decoding an n-bit opcode may comprise: a memory containing a plurality of instructions, each instruction including an n-bit opcode; a mode indicator indicating one of a plurality of execution modes; and an instruction decoder coupled to the mode indicator and the memory, the decoder decoding the opcode together with the mode indicator, wherein, for a given opcode, multiple opcode implementations exist, the instruction decoder selecting one of the multiple opcode implementations in accordance with the decoded mode indicator. The instruction decoder may decode one or more bits of the mode indicator. The instruction decoder may include a decoder input, the decoder input comprising the opcode together with said one or more bits of the mode indicator, the highest order bits of the decoder input comprising said one or more bits of the mode indicator. The implementation of one or more opcodes may comprise a hard-wired or microprogrammed implementation operating in a first of the plurality of possible execution modes, wherein the implementation of one or more other opcodes comprises a trapped implementation, the

trapped implementation executing in a second mode of the plurality of the possible execution modes.

In one embodiment, a processor having a plurality of execution modes may comprise: a memory coupled to the processor, the memory including: a plurality of instructions, each instruction comprising at least one opcode; and a mode indicator indicating a current execution mode of the plurality of execution modes, wherein for a given opcode a first opcode implementation of the instruction and a second opcode implementation of the instruction exist such that the mode indicator is operatively coupled to the memory to select one of the first opcode and the second opcode implementation.

In one embodiment, a system for executing platform-independent instructions may comprise: a processor adapted to execute instructions in a first mode and adapted to execute instructions in a second mode, the processor further adapted to trap at least one instruction associated with the first mode; a first stack associated with the first mode; a second stack associated with the second mode; one or more pointers associated with the first stack; and a trap handler emulating operations required by the at least one trapped instruction, the trap handler comprising a plurality of the instructions executable in the second mode, the processor executing the plurality of the instructions in the second mode to emulate the at least one trapped instruction, wherein the plurality of the instructions in the second mode are adapted to utilize the first stack using the one or more pointers associated with the first stack. The instruction register may be associated uniquely with the first mode, the instruction register for holding the address of an instruction executed in the first mode, wherein the plurality of the instructions in the second mode are adapted to utilize the instruction register. The one or more pointers may include a member selected from the group comprising: a stack pointer for pointing to the top of the first stack, and a local variable pointer pointing into the first stack. The system may further comprise: one or more dedicated registers, each dedicated register adapted to hold one of the one or more pointers. The first mode may comprise a Java mode and the second mode may comprise a platform dependant mode. The instructions executed in the Java mode may comprise a JVM instruction set. The instructions executed in the Java mode may comprise one or more quick instructions. The at least one trapped instruction may comprise an instruction to be quickened, wherein at least one of the plurality

of instructions adapted to access the instruction register is adapted to modify the opcode and/or operand(s) of the instruction to be quickened.

In one embodiment, a platform with which instructions are processed may comprise processing means for processing the instructions in a first mode and a second mode. The instructions executed in a first mode may comprise instructions executed in a platform independent mode, and the instructions executed in the second mode may comprise software instructions executed in a platform dependant mode. The instructions executed in a first mode may comprise software instructions compatible with a JVM compatible instruction set. The platform dependant mode may provide JVM compatible instruction set emulation for the software instructions executed in the platform dependant mode. The instructions executed in the platform dependant mode may comprise instructions compiled to a native instruction set of the platform. The second mode may comprise a C language mode.

In one embodiment, a method for processing instructions on a platform in a first mode and a second mode may comprise the steps of: providing a mode indicator to indicate the first mode or the second mode; executing the instructions in platform independent mode when the mode indicator indicates the first mode; and executing the instructions in a platform dependent mode when the mode indicator indicates the second mode. The instructions executed in a first mode may comprise instructions compatible with a JVM compatible instruction set. The platform dependant mode may a JVM compatible instruction set emulation for the software instructions executed in the platform dependant mode. The instructions executed in the platform dependant mode may comprise instructions compiled to a native instruction set of the platform. The second mode may comprise a C language mode.

In one embodiment, a multi-mode system for executing instructions may comprise: a respective stack for each mode; and a respective circuit adapted to process instructions in the respective mode using the respective stack, wherein at least one of the respective circuits uses the stack of another mode.

Further aspects, benefits, and embodiments will be apparent to those skilled in the art when viewed in light of the description and claims presented herein.

Brief Description of the Drawings

Figure 1 illustrates a conventional multi-mode processor core, such as a Java native processor;

Figure 2A illustrates details of a run time stack 140 of a conventional Java native processor;

Figure 2B illustrates a conventional stack chunk 205 comprising a portion of stack memory 140, where the stack chunking trap handler is called;

Figure 2C illustrates a memory mapping of a stack 140 during the execution of a trap handler for a trapped opcode;

Figure 3 illustrates certain aspects of a Java native processor for executing in Java and C modes;

Figures 4A and 4B illustrate details of the mode specific registers shown in Figure 3;

Figure 5 illustrates an embodiment wherein a general-purpose register file may be used as a local variable cache in a Java mode;

Figure 6A illustrates an embodiment wherein mode bit 390 is supplied as the highest order bit to a PFU look-up MEMORY;

Figure 6B illustrates an embodiment wherein mode bit 390 is supplied as a complementary microprogram MEMORY enabler;

Figure 7 illustrates the effect of certain exemplary instructions upon mode bit 390;

Figure 8 illustrates the relationship between the mode specific registers and run-time stacks of a dual mode processor;

Figure 9 illustrates an embodiment of a dual mode processor, using two stacks;

Figure 10 illustrates the use of the mode registers for implementing various trapped opcodes;

Figure 11 illustrates Java and C stacks during the execution of a trapped invokeVirtual opcode.

Description

DUAL MODE PROCESSOR

Although embodiments of the present invention are described in the context of a Java native processor wherein certain opcodes are implemented in a C mode, it is understood that

other embodiments are within the scope of the invention, which should be limited only by the claims that follow. One embodiment of the present invention may comprise a first platform independent mode and a second platform dependant mode. In one embodiment, the platform independent mode may comprise a Java mode, and the platform dependant mode may comprise an extended mode, or C mode. In one embodiment a dual mode processor may have an instruction set that may include JVM instructions as a subset thereof, with a Java mode that may be restricted to use of the JVM instructions and an extended mode that may utilize instructions other than the JVM instructions, but may include use of the JVM instructions. In other embodiments, the instructions available to the Java mode and the extended mode may be mutually exclusive.

Although an embodiment having two modes is described herein, it should be understood by those skilled in the art that more than two modes could be provided and still be within the scope of the invention described herein.

Figure 3 illustrates certain aspects of a Java native processor for executing in Java and C modes. Java native processor 300 may incorporate some of the elements of the conventional multi-mode processor, for example, PFU 301; MSU 302; microcode address input 311; microcode memory 310; microinstruction output 312; IEU 303; register file 320; flag register 321; frame pointer register 323; and a memory 330 comprising Java method space 335A including method structures 335A containing bytecode instructions (336A + 337A). In processor 300, in contrast to conventional processor 100, memory 330 comprises a Java stack 340, a C stack 380, and certain registers dedicated for exclusive use by one of the multiple modes.

MODE SPECIFIC REGISTERS

Embodiments of the present invention provide for a set of mode-specific control registers 370 for each operating mode. In the case of a Java native processor or Java accelerator having a Java mode and a C mode, a set of Java mode control registers and a set of C mode control registers are provided. In one embodiment, a mode-specific control register set 370 comprises a Java local variable pointer register 322, a Java stack pointer register 324, a Java program counter register 326, a C local variable pointer register 372, a C stack pointer register 374, and a C program counter register 376. C local variable pointer 372 points to the

base of the active C frame, while C stack pointer 374 points to the TOS of the active C frame. Similarly, while Java program counter 326 points to a memory location containing a bytecode instruction in a method structure, C program counter 376 points to an instruction in a program in C mode program space 335B. C mode program space 335B comprises a plurality of opcodes 336B and operands 337B, obtained by compiling C language program code using a C compiler specific to the instruction set of the processor core. It should be understood that other programming languages may be utilized, each language compiled using a suitable compiler for the language and target instruction set.

In one embodiment, when processor 300 is operating in C mode, C stack 380 is employed - pushing and popping C frames as C functions are called and eventually return. Likewise, when processor 300 is operating in Java mode, Java stack 340 is utilized, pushing and popping Java frames as Java methods are invoked and eventually return. In one embodiment, manipulation of C stack 380 is restricted to C mode instructions, however manipulation of Java stack 340 is permitted in both Java and C modes (i.e. both modes are "aware" of the Java stack, but only the C mode is aware of the C stack). As will be evident to those skilled in the art in light of the description that follows, these as well as other aspects of the present invention greatly simplify the operation of a processor utilizing two or more modes.

MODE BIT

In one embodiment (illustrated herein), flag register 321 includes a mode bit 390 for indicating and controlling the operational mode of the processor core 300. It should be understood that other embodiments could provide mode bit 390 in another register, or as a single latch, or other suitable storage means.

In one embodiment, C mode instructions may be provided in a first area of memory within a first memory address range, and Java mode instructions may be provided in a second area of memory within a second memory address range. For example, C mode instructions may be restricted to storage in a first region of memory, for instance, all memory locations with addresses having a '0' as the highest order bit, whereas Java code may be restricted to storage in memory addresses having a '1' as the highest order bit (i.e. a so-called method area). In one embodiment, the value of the mode bit 390 may be, therefore, implicitly defined

by the highest order bit (or other designated bit or bits) of the C mode or Java mode program counter register (PC).

In one embodiment, when a trapped opcode is to be emulated and a C frame associated with the trap handler is constructed in the C mode, flags in flag register 321 may be pushed onto the C stack 380 as return flags. Upon eventual return from the trap handler, the return flags may be popped off C stack 380 and restored to flags register 321, to reinstate the mode bit to switch execution to either Java or C mode depending upon the mode prior to the occurrence of the trap.

A similar routine may be utilized for native invokes (a C function being called from a Java method), interrupts, traps, or other similar control events that may cause a mode switch.

Where a mode switch occurs and the value of the mode bit is to change, care should be taken to preserve the correct value of the previous mode to ensure the correct mode is reinstated upon the eventual return. Consider an embodiment where the mode bit is employed either directly or indirectly for selecting the stack pointer of the correct stack, and the mode bit value of the prior mode is to be preserved on the current (i.e. new) mode's stack. The mode bit may be updated to reflect the correct value for the current mode and permit access to the corresponding stack so as to begin preserving the context of the prior mode. In so doing, the value of the prior mode would be no longer held by the mode bit; thus, writing the value of the mode bit (stored in flags register 321) to the stack would preserve the wrong mode bit value. One way to remedy this condition may include first storing the value of the flags register to a temporary memory location and subsequently pushing the contents of the temporary memory location to the stack as the return flags value. Other suitable variations may be practiced to ensure proper preservation of the prior mode bit. The temporary memory location may comprise on-chip memory such as one or more latches, registers, RAM, ROM, and the like, or suitable off-chip memory.

In one embodiment, were a mode switch is to occur, a routine is executed to implement the mode switch. The routine may comprise a microcoded routine associated with an instruction or condition that causes a mode switch, hard-wired logic associated with an instruction or condition that causes a mode switch, or a software routine executed for causing a mode switch. Such instructions and conditions may be, for example, invocation of a native

method, a trapped opcode trapping to the opcode emulating trap handler, the servicing of an interrupt, other hardware or software traps, return instructions, and the like. The routine may generally include the following steps:

1) preserve a mode indicator indicating the old mode in a temporary location;
2) change the mode indicator to the value associated with the new mode;
3) copy the temporary location to the stack associated with the new mode.

REGISTER FILE AS LOCAL VARIABLE CACHE

The general-purpose register file 327 may include GPR/LV cache control logic 328 coupled to mode bit 390 to alter read and write functionality to the general-purpose register file. In one embodiment, control logic 328 permits the general purpose register file 327 to be used as a local variable cache when operating in Java mode and a general purpose register file when operating in C mode.

REGISTER SWAPPING HARDWARE

Each pair of mode specific registers 370 may be accessed by the data processor 300. In one embodiment, each mode specific register 370 may be individually addressable. In one embodiment, instructions of the data processor's instruction set may be implemented to force a test on the mode bit or bits to determine which of the mode specific registers 370 to use. In one embodiment, two or more versions of a given instruction may be provided, one version per mode, wherein each version is implemented to access the correct mode specific register 370 for the mode of operation (e.g. C or Java mode). For instance, a first push instruction may be provided for a Java mode and second push instruction may be provided for a C mode, wherein the Java push instruction accesses the stack pointer value stored specifically in the Java stack pointer registers, and the C push instruction accesses the stack pointer value stored in the C stack pointer register.

In one embodiment, mode specific registers may be addressable as "alt-reg", and as "current-reg", representing the alternate and current mode registers, respectively, the contents of which may be defined by the value of the mode bit or bits at a given moment of execution. Figure 4A and 4B illustrate such an embodiment in the context of the mode specific registers shown in Figure 3. The "alt-reg" and "current_reg" addresses may be defined for each pair of mode specific registers 370 by providing alternate mode register 420 and current mode

register 430 for each pair of mode specific registers. Alternate mode register 420 and current mode register 430 may be tied to the outputs of a Java register 400 and a C register 410 via swap logic. The swap logic may be implemented in any manner where a "swap" signal is provided to swap the definition of alternate mode register 420 and current mode register 430 between Java register 400 and C register 410 when the data processor's mode changes.

In one embodiment, the swap signal is directly tied to the value of mode bit 390, however it is possible to provide a swap signal that is a function or derivative of the mode bit 390.

It should be understood that Java register 400 could represent any register specific to Java mode, for example, Java local variable register 322, Java stack pointer register 324, Java program counter 326, or other Java mode specific register.

Figure 4B illustrates an embodiment wherein swap logic is applied to the pairs of local variable pointer, stack pointer, and program counter registers, providing "alt_reg" and "current_reg" addresses to permit implicit reference to the appropriate mode specific register at run-time. Those skilled in the art will understand that other suitable registers may be shared via the teachings of the present invention. Corresponding C register 410 could represent any of C local variable register 372, C stack pointer register 374, or C program counter register 376. The mode specific registers may be implemented using the selection logic of Figure 4B or other similar logic.

SWAP SIGNAL

In one embodiment, a mode swap signal may be directly tied to the output of mode bit 390. C mode may, for example, be defined by a low signal on mode bit 390, and Java mode may be defined by a high value on mode bit 390. In one embodiment, to implement register-swapping functionality, a pair of multiplexers may be provided, for example, alternate multiplexer 440 and current multiplexer 450. The output of generic Java register 400 may be tied to the first input (0) of alternate multiplexer 440 and to the second (1) input of current multiplexer 450. The output of generic C register 410 may be tied to the second (1) input of alternate multiplexer 440 and to the first (0) input of current multiplexer 450. The mode swap signal, in this case the value of mode bit 390, may be supplied as the select input to alternate multiplexer 440 and current multiplexer 450. In this way, a mode bit value of 0 may

be used to select C register 410 as the value for current register 430, and Java register 400 as the value for alternate register 420. Likewise, a mode bit value of 1 may be used to select Java register 400 as the value for current register 430, and C register 410 as the value for alternate register 420.

MODE CONTROL

Execution may be changed between C and Java modes by changing the value of mode bit 390. In one embodiment, the mode may be set according to one or more bits in an instruction's address (often stored in the program counter register). Accordingly, instructions for a first mode may be stored within a first memory region, while instructions for a second mode may be stored within a second memory region such that execution of user or non-privileged code in C mode in a dual Java/C system may, thus, be prevented. By not permitting user code to explicitly include C mode instructions, system security may be increased. Because user code would then be confined to the safe Java run-time "sandbox", malicious or erroneous code may be prevented from operating on unauthorized data.

In one embodiment that utilizes a Java technology based native processor in a mobile environment, such as with a Java-enabled cellular phone, system code may be provided on ROM, and much of the system code, including programs for supporting emulated opcodes (trap handlers), interrupt service routines, and device drivers may be implemented in C mode. Such a system may be designed to constrain C mode operations to instructions stored in ROM, or at least a predetermined system area of the memory (where user code may be stored in non-volatile memory, such as FLASH memory). Accordingly, user code, such as Java MIDlets downloaded to RAM and loaded to the Java run-time system may be prevented from executing in C mode

In one embodiment, the execution of certain instructions in the processor's instruction set may cause a mode switch. Explicit mode switch instructions, wherein certain instructions serve only to switch modes, or implicit mode switch instructions, wherein certain instructions cause a mode switch while performing some other operation, may be employed to cause a mode switch. An example of the latter would include any return instruction causing execution to continue in another mode. Additionally, traps, interrupts, and exceptions may cause a mode switch. For example, in a system executing in a Java mode, the aforementioned events

may cause a handler to be called, the handler executing in C mode. In one embodiment, an implicit mode switch to Java mode may occur after the handler code returns, for instance upon restoring the flags register from the return flags value (stored on the stack), containing the mode bit(s).

ADDRESSING OF MODE SPECIFIC REGISTERS

Sets of mode specific registers 370 may also be provided without the select logic depicted in Figures 4A and 4B. For instance, where microprograms are mutually exclusive with respect to each mode, the sets of mode specific registers 370 may be accessed directly in microcode by specifically addressing each register as required.

Where certain instructions are common to both Java and C modes, and where those instructions require use of one of the mode specific registers of the current mode, the microprograms associated with the instructions may be used as "templates", similar to the concept found in C++ programming. In this manner, shared microprograms would need not account for mode in deciding which of the sets of mode specific registers 370 to use. Swap logic such as that depicted in Figures 4A and 4B may provide implicit addressing of the current register and alternate register. As such, a microprogram may be mode-agnostic and only need to address the current register and alternate register as needed with the correct value provided from the Java or C registers by swap logic. Those skilled in the art will understand that while certain aspects of the present invention are discussed in terms of "microprograms" or "microinstructions", many aspects could equally apply to hardwired embodiments.

USE OF GPRs AS LV CACHE

Although general-purpose registers 327 may provide useful benefits in a C mode, they are less useful in a stack-based mode, such as a Java mode. Because Java mode executes instructions in a stack-based manner, a large number of general-purpose registers 327 are not necessarily required in Java mode. Accordingly, in Java mode, the general-purpose registers 327 may be used for other purposes.

In one embodiment, the general-purpose registers 327 may be reused as a local variable cache in a Java mode.

In one embodiment, the general-purpose registers 327 may be reused as an extension to an existing stack cache.

In one embodiment, the general-purpose registers 327 may be reused to store at least a portion of the return execution context method frame segment.

Figure 5 illustrates an embodiment wherein the general-purpose registers 327 file may advantageously be re-used as a local variable cache in a Java mode. In one embodiment, general-purpose registers 327 may be provided with fully associative cache functionality in Java mode when driven by the mode bit 390. It is understood that the scope of the present invention as also encompasses a direct-mapped cache variant.

In one embodiment, in a fully associative cache embodiment, the values of tag registers 500 comprising a plurality of tag registers 500a to 500x may be applied as inputs to comparator 510. Depending upon the mode, address lines 505 coupled to and driven by IEU controller (not shown) may carry the address of the GPR or local variable number.

Comparator 510 may receive both the address lines 505 and tag register outputs 500a-500x. Comparator 510 may comprise a circuit for comparing address 505 with the tag register values 500a-500x. Comparator 510 may generate a register select signal to a corresponding register (327a-327x) (327n is illustrated) if the address 505 matches a tag register value. The register select signal may comprise the comparator output signal, a number of which may be provided to individually select each of the plurality of general-purpose registers 327a-327x. If no corresponding tag value 500 is found, a miss signal (not shown) may be generated, signaling that the desired local variable need be fetched from memory, or the next level of cache, for instance, if a general data or instruction and data cache is provided. Address lines 505 may also be coupled to a decoder for decoding address 505 into a select signal corresponding to one of registers 327a-327x. A register select multiplexer 530 may be provided for each register 327a-327x for selecting either the output of decoder 520 or comparator 510. The selection may be determined by the mode bit (signal) 390, applied as a select signal to each register select multiplexer 530. Driven by the mode signal 390, a plurality of general-purpose registers 327a-327x may serve as a local variable cache. Accordingly, the hardware provided for the general-purpose registers may be reused as a local variable cache for use in Java mode and additional hardware need not be provided.

MODE BIT AS uPROGRAM ADDRESS BIT [or complementary uROM enabler]

Hardware implementations of standardized platforms, such as the JVM, may constrain architecture design in certain ways. One constraint may be imposed by opcode size. For instance, the JVM specification provides for byte-size (8-bit) opcodes. Consequently, the JVM instruction set is limited to no more than 256 (2^8) opcodes. In one embodiment, JVM may specify less than 256 opcodes. Thus, a portion of the 256 possible opcodes may be used to implement the functionality of embodiments described herein. In one embodiment, where a second mode is provided in addition to a first mode based upon a standardized platform, it may be desirable to provide additional instructions, beyond what is permitted by the opcode size defined by the standardized platform. However, without changing the size of the platform's opcode, design of such a system becomes a complex task, for example, a second decoder circuit may be required.

In picoJava processors, the issue of a limited number opcodes is addressed by providing a so-called "escape opcode" to extend the instruction set. One of the 256 opcodes is designated as an escape opcode. The escape opcode serves to indicate that the next byte in memory is an opcode of an extended instruction set. Thus, opcode 0x60h would be decoded as an iadd (integer addition) instruction in a first Java mode with this opcode re-used to designate and extend the 256 opcodes by preceding it with an escape opcode. Unfortunately, this is somewhat inefficient to process and wastes an opcode for the escape opcode. Additionally, if more than two modes were provided with each mode having its own instruction set, or where additional instructions are required, further opcodes would have to be wasted in order to provide multiple escape opcodes.

In contrast to other approaches such as picoJava, one embodiment of the present invention uses a mode bit or bits to extend the number of instructions in a limited instruction set size, for example, a standardized instruction set. In one embodiment, a mode bit or bits may be supplied as the as the highest order bits of an operand to increase the number of opcode values. In this way, the number of opcodes may be increased using only a single decoder arrangement without jeopardizing or complicating the implementation of a standardized instruction set.

Additionally, where microprograms of instructions for each mode may be divided by mode across multiple microcode memories, the mode bit or bits may be utilized as a

complementary enable signal to the microcode memories. In this way, only one of the microcode memories is enabled and powered-up for access at a time. In one embodiment, power savings may be on the order of 5-25% depending on the memory technology used (e.g. gates or RAM).

In one embodiment, a mode bit may be supplied to a decoder means as the highest order bit of an opcode. Any suitable decoder means may be used such as gates, RAM, ROM, PLA, PAL, and the like. In one embodiment, the decoder means may comprise a microprogram address look-up table in a pre-fetch unit. In one embodiment, two microprogram memories may be provided and the mode bit utilized as a complimentary enable bit to each memory.

Figure 6A illustrates an embodiment wherein a mode bit 390 is supplied as the highest order bit to an instruction decoder 600 in PFU 101. PFU 101 fetches instructions from system memory 130. An instruction may be of variable length and may comprise an opcode and possibly one or more operands. PFU 101 may isolate the opcodes from the stream of fetched instruction data and may supply opcodes to instruction decoder 600. Instruction decoder 600 may comprise a hardwired decoder, a look-up table memory, or other suitable decoding arrangement. Where a look-up table is utilized, the opcodes may serve as an address into the look-up table to identify a microprogram address 111 corresponding to a given opcode. Microprogram address 111 may be supplied to microprogram memory 110 to identify the microprogram that implements the opcode.

Using a single mode bit 390 (or a swap signal, similar to the above-described swap signal) as the highest order bit of an opcode, twice as many opcodes may be uniquely identified than otherwise permitted using the opcode size defined by a standard. Furthermore, this embodiment may advantageously be implemented using only one decoder circuit.

In the embodiment illustrated in Figure 6A, 8 bit opcodes (bytecodes) are supplied to the instruction decoder 600. Mode bit 390 is also supplied to instruction decoder 600 as the highest order bit of the opcode, thereby providing a 9-bit opcode. In this way, 512 unique opcodes may be identified.

In one embodiment, where an extended mode opcode requires the same functionality as a Java opcode, combinatorial logic may be supplied to modify the mode bit PFU input to

map out to the Java opcode. Although this may complicate hardware design and decrease the flexibility and the total number of opcodes that may be used, microprogram development may be simplified because only one instance of a microprogram need be created and debugged. In one embodiment, multiple opcodes of two or more modes may decode into a common microprogram address, thereby sharing microcode between modes and simplifying microprogram development and possibly the size of the microprogram memory.

Referring again to Figure 6A, instruction decoder inputs of 0xxxxxxxx may correspond to one of 256 Java mode instructions, including those specified by the JVM specification and possible Java optimization opcodes such as certain quickened variants and common instruction sequence folding opcodes. Likewise, instruction decoder inputs of 1xxxxxxxx may correspond to one of 256 extended mode instructions. At least one of the Java mode instructions may comprise an implicit or explicit mode switch instruction. In one embodiment, a Java mode trap instruction may provide for all software and hardware traps. Extended mode instructions may include at least one return instruction to change modes back to Java mode. In a multi-stage, pipelined processor embodiment, opcodes in stages between an opcode decoding stage and an execution stage of a mode switching opcode may incorrectly interpret opcodes when a mode switching opcode is executed, the mode switching opcodes may comprise opcodes that would cause a pipeline flush anyways (e.g. invoke and return instructions). Thus the incorrectly interpreted opcodes would not reach the execution stage where they could incorrectly alter data or the state of the processor.

Figure 6B illustrates an alternate embodiment wherein mode bit 390 is supplied as a complementary microcode memory enabler. In one embodiment, MSU 110 may comprise two memories (e.g. ROM, RAM), a first microcode memory 620 for storing C mode microprograms, and a second microcode memory 615 for storing Java mode microprograms. Microprogram addresses 111 may be supplied as inputs to both microcode memories. Microcode memories 615 and 620 may also include enable (or select) inputs for enabling the memories upon assertion of a predetermined signal.

Mode bit 390 may be supplied to both enable inputs of the microcode memories, including an inverter on the enable input to the first microcode memory. In this way, mode

bit 390 may enable only the appropriate microprogram memory, depending upon the current mode.

In one embodiment, extended mode instructions may comprise one or more opcodes for providing register-based operations. Register-based operations may include operations that operate upon data stored in specific registers, as opposed to stack-based operations that operate on data stored on a stack. Register-based instructions may specify a source and/or destination register as operand(s).

INSTRUCTIONS & CONDITIONS HAVING EFFECT ON MODE BIT

Figure 7 illustrates the effect of a few representative instructions upon mode bit 390. Mode bit 390 may be altered during the execution of one or more instructions or upon the occurrence of certain conditions. Such mode changing instructions or conditions may be characterized as either J-switch, causing a switch to Java mode, or C-switch, causing a switch to C mode. Referring to Figure 7, C-switch instructions and conditions are generally represented by 700. Each C-switch instruction and condition 700 may cause a change in mode bit 390 to represent the C mode state (the value of which is predetermined). J-switch instructions and conditions are generally represented by 710. Each J-switch instruction and condition 710 may cause a change in mode bit 390 to represent the Java mode state.

The execution of certain instructions may cause a change in mode by writing the appropriate value to the mode bit. For example, in one embodiment, a Java invoke instruction may be implemented in microcode (such as an invoke_quick instruction). If the method being invoked is flagged as a native method, or the opcode is specifically a native invoke opcode, the execution of the invoke instruction may switch the mode from Java to C mode.

In embodiments where traps handlers are developed in C code, the trap handlers are executed in C mode. Accordingly, the occurrence of a trap would be considered a C-switch condition. Where certain Java mode opcodes are emulated via trap handlers in C mode, the execution of those opcodes may likewise cause a change of mode from Java to C mode.

In one embodiment, interrupt service routines may be developed in C mode so they may be compiled in a platform-dependant manner to access extended mode instructions. Consequently, the occurrence of an interrupt may cause the mode to change to C mode if the interrupt occurred while executing in Java mode.

In one embodiment, handlers for both traps and interrupts may be mapped in a single vector table. The opcode emulation traps may be mapped to the opcode's number (i.e. opcode 0x0A corresponds to the tenth entry in the vector table), the trap table entries having numbers matching non-trapped opcodes being utilized for interrupt handler addresses.

Each of the above-described causes of mode switches may have corresponding return instructions that switch the mode back to Java when ultimately executed. For instance, the C-code implementation of a native method (called from Java mode via invoke native) may be compiled using a platform-dependant compiler, the compiler inserting a return instruction that causes a mode switch back to the caller in Java mode. In one embodiment the mode switch is caused by popping a return flags value containing the saved mode bit state, from the trap frame (on the C stack) corresponding to the native method. In one embodiment, one or more return instructions specific to native methods may be provided, the return instructions explicitly changing the mode bit. Those skilled in the art will understand that other mechanisms may be utilized to cause the mode switch.

C-to-C FUNCTION CALLS

C mode function calls (i.e. a C function call while executing in C mode) and their corresponding returns do not necessarily need to modify the mode bit 390. Using a platform-dependant compiler, C language code may compile to a C-to-C function call instruction that may specify the address of a function in C code program space in memory. The microcode implementation of C-to-C call instruction may build a variant of the trap frame wherein the return flags are not included. This is possible as no mode switch occurs on the call/return sequence on a C-to-C call/return. Thus, the saved mode bit state in the return flags is not required to properly return to the caller. The platform-dependant compiler may compile the C function's return instruction to a C-to-C return instruction. When executed in the processor core, the microcode (or hardwired) implementation of this instruction would not attempt to pop a return flag off the C stack. It should be noted that the same return instruction (that includes popping the return flags value from the stack) could also be used, but one would need to ensure that the flags register is pushed when the C frame is constructed on the stack.

USE OF ALT REGISTERS IN C MODE / TWO STACKS WITH ALT_REGs

In one embodiment, access to various components of the Java mode execution context, while executing in the C mode, is facilitated. Referring to Figure 8, a Java stack 340 and C stack 380 are illustrated at some point in execution of a program, each stack containing a number of Java frames 850 and C frames 860, respectively. A call chain of several Java methods is represented on the Java stack 340. During the execution of a bytecode in the Java context associated with Java frame 850, a trap, interrupt, or the like, may occur. In Figure 8, C mode execution of a trap or interrupt handler is initiated, resulting in the construction of first C frame 860A. At some point in its execution, the handler issues a function call to perform some operation, resulting in second C frame 860B. The function associated with second C frame 860B issues a call to another function, resulting in the construction of a third C frame 860C on C stack 380. It is in the context of this function that the current execution "snap-shot" of Figure 8 is taken. The top Java frame 850 and C frames 860A, 860B and 860C are illustrated in detail.

The top Java frame 850 comprises local variable segment 851, return execution context segment 852, and local stack segment 853. The Java local variable pointer 322 points to local variable segment 851. Java stack pointer 324 points to the TOS element in local stack segment 853. Java program counter 326 points to the current opcode being executed in Java mode.

The top (third) C frame 860C comprises return PC 861C, return local variable pointer 862C, and local stack 863C. C local variable pointer 372 points to the return local variable pointer 862. C stack pointer 374 points to the TOS element in local stack segment 863C. C program counter 376 points to the current instruction in the C function being executed. Third C frame 860C is an example of a C frame constructed as a result of a function call from C mode (i.e. a C function called from another C function).

Second C frame 860B is similar to top C frame 860C, however mode specific pointers do not point into it as second C frame 860B is not the active frame.

First C frame 860A differs from C frames 860B and 860C. First C frame is constructed as a result of a mode C-switch condition or instruction, thus the frame includes the return flags value 864. The return flags 864 contain the return mode bit, in this case holding a value associated with Java mode. In this way, when execution returns to the context

associated with first C frame 860A, execution of the proper return instruction may pop return flags 864 from C stack 380 and write the value to the flags register, causing a mode switch back to Java mode.

PREFERRED STACK DESIGN (CHUNKING IN JAVA; STATIC IN C)

Figure 9 illustrates one embodiment of a dual mode processor using two stacks. The embodiment illustrated in Figure 9 is similar to that of Figure 8, however in this embodiment Java stack 940 comprises a stack-chunked system.

Although stack chunking is known, its application to dual mode processors, such as Java technology-based native processors and devices as described herein is not. For example, although picoJava utilizes stack chunking on its run-time stack, C mode operations are conducted on the same stack as Java mode operations, and C mode operations are subject to the overhead and complexities associated with stack chunking.

In the illustrated embodiment of Figure 9, Java stack 940 comprises first stack chunk 900A and second stack chunk 900B. First and second stack chunks 900A and 900B are linked by methods known in the stack chunking art. Figure 9 suggests an implementation of stack chunks using stack chunk data structures that include a stack chunk data structure header 930, shown on first stack chunk 900A (not shown on stack chunk 900B to simplify the figure). Stack chunk data structure 930 may comprise one or more pointers, one of which may point to another (next) stack chunk. As illustrated, a pointer in stack chunk data structure header 930 points to second stack chunk 900B, indicating that this is the "next" stack chunk. A stack limit pointer is also maintained, for instance, in a stack limit register 920.

In this particular example, similar to the example illustrated in Figure 8, a Java call chain may be developed on Java stack 940. The Java call chain, comprising Java frames 950A, 950B, 950C, 950D, 950E, 950F, 950G and 950H, may exceed the memory allocated to first stack chunk 900A. During the invocation of the Java method associated with Java frame 950F, by checking against the value of stack limit 920, it may be determined that insufficient memory remains in first stack chunk 900A to accommodate Java frame 950F. At this point in execution, a trap may occur and a stack chunk trap handler may be executed, resulting in a call to a memory allocation routine. A new stack chunk may be allocated, providing second stack chunk 900B the proper linkages being established in stack chunk data structure header

930. Java mode may continue execution on second stack chunk 900B, resulting in the construction of Java frames 950F, 950G, and 950H. As shown, Java frame 950H is the top Java frame, indicated by Java local variable pointer and Java stack pointer. A Java frame pointer may also be provided for the express purpose of explicitly indicating the current Java frame.

In this example, a trap or interrupt occurs at some point in the context associated with top Java frame 950H. As in the previous example in Figure 8, instead of executing the trap handler in the stack chunked domain (the Java stack), the trap or interrupt handler may execute on C stack 380. An important difference to note in this example is that, while Java stack 940 memory is stack chunked, C stack 380 memory is statically allocated.

It is identified that a mix of these two stack allocation strategies (stack chunking on Java stack and static allocation on C stack) yields a balanced approach for each mode of execution. In providing each mode of execution with its own stack, memory allocation for each mode's stack may utilize the approach that best suits that mode.

In one embodiment, the Java mode, in which stack memory requirement information is available for each method, may efficiently make use of a stack chunking routine by only checking the stack limit on method invocations (and not on every data push step).

On the other hand, C mode, in which stack memory requirement information may not be available for each function, would necessitate checking the stack limit on every push of data to the stack. This may reduce the performance of C mode execution. Furthermore, whenever a new stack chunk is allocated, or, using prior art stack chunking methods, whenever a return instruction causes execution to cross a stack chunk boundary, stack chunk handling routines may interrupt the program flow at unpredictable moments.

Accordingly, in one embodiment, C stack 380 may be statically allocated, while Java stack 340 is stack chunked to provide an optimal balance of high performance and aggressive memory conservation. This embodiment also provides the benefit of reduced maximum interrupt latency. Because interrupt service routines generally execute in C mode (with C mode executing on statically allocated C stack 380) data push operations are faster and there is no risk of a stack chunk trap occurring. The latter is of particular concern because an ISR that causes the stack limit to be exceeded, thus, incurring a performance penalty of trapping to

a stack chunking trap, would take a double performance hit as it may also cause a stack chunk trap on its return when execution crosses back to the old stack chunk. Furthermore, the use of prior art stack chunk trap handlers may require any elements of the stack frame (associated with the offending context) already stored on the old stack chunk to be copied to the new stack chunk in order to maintain stack frame integrity. This increases the penalty incurred by stack chunking traps.

The size of a statically allocated C stack 380 may be determined by those skilled in the art. In some embodiments, static memory allocation may be user programmable. Static memory allocation is well known in the art.

EXAMPLES OF USES AND ADVANTAGES OF THE ALT_REGS

Figure 10 illustrates the use of the mode specific registers for implementing various trapped (or emulated) opcodes. In one embodiment, when operating in C mode, the C stack may be utilized for most operations. However, in the case of a trapped opcode, the C mode emulates a Java mode opcode operation. Accordingly, C mode may need to be aware of the Java stack and registers to read from and write to the top Java frame on the Java stack. Several examples of C mode-based Java manipulations are described below. Figure 10 depicts an example Java frame 1050 associated with a Java execution context in which a trapped opcode 1036 is encountered.

JAVA/ALT LOCAL VARIABLE POINTER

In one embodiment, emulated operations requiring access to local variables in the local variable segment 1051 of Java frame 1050 may access the local variables (such as local variables 1051A, 1051B, and 1051C), via Java LP register 322. Alternatively, where a mode specific register swapping mechanism (such as that illustrated in Figure 4A and 4B) is provided, alt_LP register may be used, as the alt_LP register would automatically correspond to the Java LP while executing in C mode. In any event, rapid access to the Java local variables is provided by embodiments of the present invention.

Examples of operations that may be conducted on Java local variables from C mode as facilitated by embodiments of the present invention include load and store operations, access to parameters passed to a method, and access to the "this" reference (typically the first (LV[0]) local variable, received as a parameter, of non-static method. The first local variable

1051A may be accessed directly using Java local variable pointer 322. An index may be applied to Java local variable pointer 322 to access any of the other local variables. For example, second local variable 1051B (LV[1]) may be accessed using an index of one, while third local variable (LV[2]) may be accessed using an index of two.

JAVA/ALT STACK POINTER

In one embodiment, the local stack operands stored in local stack segment 1053 may be accessed from C mode via the Java SP register 324. Java mode stack operands 1053 may be pushed or popped as required to properly emulate a Java bytecode using a C mode trap handler. Likewise, in embodiments where the mode specific register swapping mechanism is provided, alt_SP register may be used, as the alt_SP register would correspond to the Java SP while executing in C mode.

By way of example, Java SP may be used for the emulation of floating point arithmetic opcodes in a Java native processor that does not include a floating-point unit (FPU). Floating point opcodes such as fmul, fadd, fsub, and the like may be implemented using trapped opcodes. Assuming Java stack 340 is 32 bits wide, each of the aforementioned floating point instructions would pop (read) the top four stack operands from local stack operand segment 1053, perform the appropriate operations to generate a result, and push (write) two words comprising the result to the local stack operand segment. In one embodiment, regardless of how deep a C mode call chain may be on C stack 380, access to the required Java stack operands may be performed with O(1) behavior by accessing Java stack pointer register 322 or the alternate stack pointer register.

JAVA/ALT PROGRAM COUNTER

In one embodiment, where access to the Java opcode 1036 may be required when emulating certain Java opcodes in C mode, the alternate program counter (alt-PC) may be used to access the operands of a processor. One important use of the alternate program counter may be to implement self-modifying Java mode code in C mode. One Java optimization known in the art is the use of so-called "quickening". Many Java operations require following several levels of indirection into a "constant pool", a self-referential structure that facilitates dynamic linking. Entering the constant pool is time consuming and reduces system performance. Quickening provides a method of saving the results obtained

from a first execution of a constant pool operation so that that the constant pool need not be entered should that very same instruction be subsequently encountered. One method of quickening involves replacement of Java opcode 1036 and/or Java operands 1037A, 1037B in memory. Because instructions that need to enter the constant pool tend to be implemented as trapped opcodes, the trap handler may include code to quicken the instruction. To do so requires access to the Java program counter. In one embodiment, access to the Java program counter may be efficiently achieved by either accessing the Java program counter directly, or the alternate program counter (as the trap operates in C mode).

RUN-TIME EXAMPLE – TRAPPED INVOKEVIRTUAL

Figure 11 illustrates some of the stack operations that may be conducted during the execution of an invoke virtual opcode.

Although the details of the operations required to implement an invokeVirtual opcode are well known in the art, some of the steps involved in an invoke virtual are detailed below to illustrate some of the benefits of the present invention.

As illustrated in Figure 11 caller associated with Java frame 1100 on Java stack 340 may push onto Java stack 340 an object reference 1101 and two parameters: first parameter 1102 and second parameter 1103. Instance (i.e. non-static) methods access the instance, or object, whose method is invoked, via object reference 1101. As the invokeVirtual opcode in this example is trapped and consequently implemented by a C mode trap handler, the Java or alt stack pointer register may be used to access object reference 1101. The operand provided in the invokeVirtual instruction includes a constant pool reference. The constant pool reference is used to identify the method (through steps commonly known as resolution) and find the method entry data structure associated with the method. Once the method entry is found in memory, information related to the method to be invoked may be accessed.

The stack requirements of the method being invoked may be determined from data available in a method entry data structure. The invokeVirtual opcode may use this information to compare the maximum stack requirements of the method with the amount of available stack memory in the current stack chunk. To do this, the Java stack pointer is accessed. If sufficient stack memory remains, execution continues, otherwise a trap occurs to handle the stack chunking operations.

Java stack pointer 324 may be accessed again and may be incremented, as needed, to provide room in local variable segment 1120 for new local variables 1121 (that will be stored during method execution between the existing parameters and the return execution context segment 1125).

As further illustrated, return execution context 1125 may be constructed, pushing the requisite data on Java stack 340 at the location determined by the Java stack pointer 324, incrementing Java stack pointer 324 for every data element pushed thereon. While the exact contents of return execution context 1125 may vary between implementations, data such as the return Java program register 326, return Java local variable pointer 322, and the like, may be commonly included. Prior to being pushed into return execution context 1125, the Java program register 326 may be changed to point to the instruction following the invokeVirtual instruction.

The various registers may be updated to the context of the new (invoked) method, the Java local variable pointer being modified to point to the first local variable (object reference). Java stack pointer 324 holds the correct value as it is updated with the push of every return execution context 1125 element. Java program counter 326 is also updated to point to the first bytecode of the method (available through the method entry data structure).

Quickening may optionally be performed. The invokeVirtual trap handler may access the invokeVirtual opcode via Java program counter 326 to thereby replace the invokeVirtual opcode with a quickened variant of the opcode (such as invokeVirutalQuick). Depending upon the quickening mechanism, either the operand or a constant pool entry may be also modified. In the case of the former, the operand (the Java Virtual Machine Specification teaches a two- byte operand for the constant pool reference) may be replaced with the results of the method resolution. In one embodiment, the two-byte opcode may be replaced with a reference into a method table corresponding to a reference to the appropriate method entry, and the number of parameters (or arguments) taken by the method. Other variants of the quickening process are also possible.

The invokeVirtual trap handler, operating in C mode, may execute the appropriate return instruction, may pop the trap handler's frame off the C stack (not shown) and may transfer control to Java mode to be ready to execute the first bytecode of the invoked method.

Accordingly, the mode specific registers and multi-stack embodiments of the present invention are clearly advantageous in the execution of the embodiments disclosed herein.

Numerous modifications and variations of the present invention are possible in light of the above embodiments described herein, for example, using other suitable computer readable medium to implement one or more elements of the embodiments discussed herein. A computer readable medium is any medium known in the art capable of storing information. Computer readable media include Read Only Memory (ROM), Random Access Memory (RAM), flash memory, Erasable-Programmable Read Only Memory (EPROM), non-volatile random access memory, memory-stick, magnetic disk drive, floppy disk drive, compact-disk read-only-memory (CD-ROM) drive, transistor-based memory or other computer-readable memory devices as is known in the art for storing and retrieving data, for example such as SRAM, latches, flip-flops, and the like. Furthermore, other languages are within the scope of those described herein, including Java 2 Standard Edition (J2SE), Java 2 Micro Edition (J2ME), and configurations such as Connected Limited Device Configuration (CLDC) or Connected Device Configuration (CDC) available from Sun Microsystems, Inc., Palo Alto, California; and other Java or Java-like languages, for example, Common Language Interchange (CLI), Intermediate Language (IL) and Common Language Run-time (CLR) environments, and C# programming language which are part of the .NET and .NET Compact framework available from Microsoft Corporation Redmond, Washington; and Binary Run-time Environment for Wireless (BREW) from Qualcomm Inc., San Diego, California. It is therefore to be understood that within the scope of the appended claims, the invention may be practiced otherwise than as specifically described herein.

What is Claimed is:

1. A dual-mode system for executing instructions, comprising:

   a first stack and a second stack;

   a circuit adapted to execute instructions in a first mode using the first stack; and

   a circuit adapted to execute instructions in a second mode using the first stack and the second stack.

2. The system of Claim 1, wherein the first and second stack are adapted to store information.

3. The system of Claim 2, wherein

   instructions executed in the first mode operate on information stored on the first stack, and

   instructions executed in the second mode operate on information stored on the first stack and the second stack.

4. The system of Claim 3, wherein the circuit adapted to execute instructions in a first mode and the circuit adapted to execute instructions in a second mode comprises a data processor.

5. The system of Claim 3, wherein the circuit adapted to execute instructions in a first mode, and wherein the circuit adapted to execute instructions in a second mode comprises a data processor coupled to a hardware instruction accelerator.

6. The system of Claim 5, wherein the hardware instruction translator comprises an instruction accelerator adapted to utilize a JVM instruction set.

7. The system of Claim 4, wherein the data processor comprises a Java technology-based native processor.

8. The system of Claim 6 or 7 further comprising one or more memories, the memories operatively coupled to the circuit adapted to execute instructions in a first mode and to the circuit adapted to execute instructions in a second mode, the memories storing the first stack and the second stack.

9. The system of Claim 1, wherein the first stack comprises one or more stack chunks.

10. The system of Claim 9, wherein the second stack comprises a statically allocated stack.

11. The system of Claim 10, wherein the instructions executed in the first mode comprise software instructions executed in a platform independent mode, and wherein the instructions executed in the second mode comprise software instructions executed in a platform dependant mode.

12. The system of Claim 10, wherein the instructions executed in the first mode comprise software instructions executed in a Java mode, and wherein the instructions executed in the second mode comprise software instructions executed in a platform dependant mode

13. The system of Claim 12, wherein the platform dependant mode provides emulation of at least one software instruction of the Java mode.

14. The system of Claim 13, wherein the platform dependant mode comprises a C mode, the instructions executed in the C mode comprising instructions compiled from one or more C language functions to a native instruction set of the system.

15. The system of Claim 13, wherein the instructions executed in the first mode comprise object-oriented software instructions executed in an object-oriented mode, and wherein the instructions executed in the second mode comprise software instructions executed in a platform dependant mode, the platform dependant mode providing object-oriented software instruction emulation.

16. The system of Claim 8, the one or more memories defining

one or more memory locations for use in a first mode and storing information related to an execution context of the first mode,

one or more memory locations for use in a second mode and storing information related to an execution context of the second mode, and

the one or more memory locations for use in a first mode being available and addressable to software instructions operating in the second mode.

17. The system of Claim 16, the one or more memory locations for use in a first mode and the one or more memory locations for use in a second mode storing information

comprising one or more values selected from a program counter, a stack pointer, a local variable pointer, and a frame pointer.

18. The system of Claim 16, the one or more memory locations for use in a first mode comprising one or more registers, and the one or more memory locations for use in a second mode comprising one or more registers.

19. The system of Claim 16, wherein the one or more memories defines a mode indicator for indicating and controlling whether the system executes instructions in the first mode or in the second mode.

20. The system according to Claim 1 further comprising a mode indicator adapted to control which of the circuit adapted to execute instructions in a first mode and the circuit adapted to execute instructions in a second mode is active, the mode indicator having a first state indicating the circuit adapted to execute instructions in a first mode is active and a second state indicating the circuit adapted to execute instructions in a second mode is active.

21. The system according to Claim 19, wherein access to the one or more memory locations for use in a first mode and access to the one or more memory locations for use in a second mode is controlled by a status of the mode indicator.

22. The system according to Claim 21 further adapted to maintain a prior mode indicator history such that after completion of an operation in a current mode a prior mode can be restored.

23. The system according to Claim 22, wherein the prior mode indicator history comprises a return mode indicator.

24. The system according to Claim 23, wherein the return mode indicator is stored in an element in a return execution context on one of the first stack and the second stack, and wherein the one of the first stack and the second stack is selected by the prior mode indicator.

25. The system according to Claim 24, wherein the element storing the return mode indicator comprises a return flags element.

26. The system according to Claim 24, wherein the element in a return execution context includes one or more bits for representing the return mode indicator.

27. The system according to Claim 26, wherein the operation comprises the execution of a return instruction.

28. The system according to Claim 24, further comprising:

a mode switch control adapted to set the mode indicator to the first state upon occurrence of any one of a first set of conditions and adapted to set the mode indicator to the second state upon occurrence of any one of a second set of conditions.

29. The system according to Claim 28, wherein the first set of conditions comprises one or more of the group comprising: a software trap; a hardware trap; the servicing of an interrupt; while in Java mode, a native method call; and an explicit software controlled change of the state of the mode indicator to set the state to the second state.

30. The system according to Claim 28, wherein the second set of conditions comprises one or more of the group comprising: the execution of a return instruction, the return instruction returning from a trap handler; the execution of a return instruction, the return instruction returning from an interrupt handler; the execution of a return instruction, the return instruction returning from a native method, the native method previously called from an instruction executed in the first mode; and an explicit software controlled change of the state of the mode indicator to set the state to the second state.

31. The system according to Claim 29, wherein the first set of conditions comprises a hardware trap, wherein the hardware trap comprises a trap to an instruction emulation trap, the instruction emulation trap emulating an instruction selected from the group comprising: an object-oriented instruction, a platform independent instruction, and a Java instruction.

32. A system including a processor having multiple execution modes, the arrangement comprising:

a mode indicator adapted to indicate one of a plurality of execution modes; and

a memory storing at least one set of values, each set of values having a respective member for each mode for each of the plurality of execution modes,

wherein the mode indicator is further adapted to select the respective member for the one of the plurality of execution modes for each set of values.

33. A system according to Claim 20 further comprising:

a first set of registers specific to the first mode; and

a second set of registers specific to the second mode;

wherein the first set of registers is addressable when said mode indicator is in the first state, and the second set of registers is addressable when said mode indicator is in the second state, and wherein the first set of registers is addressable when said mode indicator is in the second state.

34. A system according to Claim 33 further comprising multiplexing circuitry adapted to select either the first set of registers or the second set of registers as a function of the mode indicator.

35. The system according to Claim 34, further comprising:

a set of current registers coupled to the output of the multiplexing circuitry; and

a set of alternate registers coupled to the output of the multiplexing circuitry,

the multiplexing circuitry selecting the first set of registers as the content of the current registers when the mode indicator is set to indicate the first state,

the multiplexing circuitry selecting the second set of registers as the content of the alternate registers when the mode indicator is set to indicate the first state;

the multiplexing circuitry selecting the second set of registers as the content of the current registers when the mode indicator is set to indicate the second state, and,

the multiplexing circuitry selecting the first set of registers as the content of the alternate registers when the mode indicator is set to indicate the second state.

36. The system according to Claim 35 further comprising:

at least one instruction which is used in both the first and second mode which when executed is adapted to access one or more of the registers in the set of current registers.

37. A dual-mode system for executing instructions comprising:

at least one set of registers, each set of registers being used as a cache in a first mode and as a general-purpose register in a second mode.

38. The system according to Claim 1 further comprising:

a set of registers, the set of registers capable of being used as a cache in the first mode and as a general purpose register in the second mode.

39. The system according to Claims 37 or 38 wherein the cache comprises a local variable cache.

40. The system according to Claim 39 wherein the cache comprises a Java local variable cache.

41. The system according to Claim 40 wherein the Java local variable cache is a direct-mapped cache or an associative cache.

42. An instruction decoder adapted to employ one or more mode bits supplied as the highest order bits of a decoder input in combination with fixed length opcodes to increase the number of possible opcode values.

43. A multi-mode system for executing instructions, comprising:

an instruction fetch unit for receiving instructions;

a mode indicator

adapted to indicate one of the plurality of modes, the mode indicator comprising one or more bits;

an instruction decoder

coupled to the instruction fetch unit and

adapted to the decode the instructions as an n-bit opcode,

the instruction decoder comprising an input adapted to receive the n-bit opcode and the mode indicator,

the one or more bits of the mode indicator comprising one or more highest order bits of the decoder input; and

a processor

coupled to the instruction fetch unit and

adapted to perform a first set of one or more operations when an opcode of an instruction is decoded in the first mode, and

adapted to perform a second set of one or more operations when an opcode of an instruction is decoded in the second mode.

44. The system according to Claim 43 wherein the instruction decoder is selected from a group comprising: a software decoder, a hard-wired logic circuit decoder, and a memory-based decoder.

45. The system of Claim 44 wherein the instruction decoder comprise a hard-wired logic circuit decoder, the hard-wired logic circuit decoder selected from the group comprising: a PLA circuit; a PAL circuit; and a circuit, the circuit comprising one or more gates.

46. The system of Claim 44 wherein the instruction decoder comprises a memory-based decoder, the memory-based decoder selected from the group comprising: a ROM circuit storing a look-up table, a RAM circuit storing a look-up table, and a Flash memory circuit storing a look-up table.

47. The system of Claim 44, wherein the instruction decoder is integral with the processor.

48. The system of Claim 47, wherein the processor comprises a Java technology-based native processor.

49. The system of Claim 47, wherein the processor comprises an object-oriented processor.

50. The system of Claim 46, wherein the instruction decoder comprises a Java technology-based accelerator circuit.

51. The system of Claim 44, wherein the first set of one or more operations comprises one opcode of a first instruction set, and the second set of one or more operations comprises one opcode of a second instruction set.

52. The system of Claim 44, wherein the plurality of modes comprises two modes, the mode indicator comprising one mode bit, the decoder decoding up to 2^{n+1} possible opcodes, wherein the first instruction set comprises one or more of the first 2^n opcodes of the 2^{n+1} possible opcodes, and wherein the second instruction set comprises one or more of the 2^n+1 to 2^{n+1} opcodes of the 2^{n+1} possible opcodes.

53. The system of Claim 52, wherein n is equal to 8.

54. The system of Claim 52, wherein the second mode comprises a platform dependant mode and the second instruction set comprises a platform dependant instruction set.

55. The system of Claim 54, wherein the first mode comprises a Java mode and the first instruction set comprises an instruction set specified in the Java Virtual Machine specification.

56. The system of Claim 55, wherein the first instruction set further comprises one or more quick instructions.

57. The system of Claim 55, wherein the first instruction set further comprises one or more folded instructions.

58. The system of Claim 54, wherein the platform dependant instruction set comprises one or more opcodes for accessing specific memory locations.

59. The processor of Claim 54, wherein the platform dependant instruction set comprises one or more opcodes for providing register-based operations.

60. The processor of Claim 58, wherein the platform dependant mode comprises a C mode, and wherein the instructions executed in the platform dependant mode comprise instructions compiled from the C programming language to one or more instructions of the platform dependant instruction set.

61. A multi-mode system for decoding an n-bit opcode comprising:

a memory containing a plurality of instructions, each instruction including an n-bit opcode;

a mode indicator indicating one of a plurality of execution modes; and

an instruction decoder coupled to the mode indicator and the memory, the decoder decoding the opcode together with the mode indicator,

wherein,

for a given opcode, multiple opcode implementations exist,

the instruction decoder selecting one of the multiple opcode implementations in accordance with the decoded mode indicator.

62. The multi-mode system of Claim 61, wherein the instruction decoder decodes one or more bits of the mode indicator.

63. The multi-mode system of Claim 62, wherein the instruction decoder includes a decoder input, the decoder input operatively coupled to the opcode together with said one or more bits of the mode indicator, the highest order bits of the decoder input comprising said one or more bits of the mode indicator.

64. The processor of Claim 61, wherein

the implementation of one or more opcodes comprises a hard-wired or microprogrammed implementation operating in a first of the plurality of possible execution modes, and

wherein the implementation of one or more other opcodes comprises a trapped implementation, the trapped implementation executing in a second mode of the plurality of the possible execution modes.

65. A processor having a plurality of execution modes, the processor comprising:

a memory coupled to the processor, the memory including:

a plurality of instructions, each instruction comprising at least one opcode; and

a mode indicator indicating a current execution mode of the plurality of execution modes,

wherein for a given opcode

a first opcode implementation of the instruction and a second opcode implementation of the instruction exist such that the mode indicator is operatively coupled to the memory to select one of the first opcode and the second opcode implementation.

66. A system for executing platform-independent instructions comprising:

a processor adapted to execute instructions in a first mode and adapted to execute instructions in a second mode, the processor further adapted to trap at least one instruction associated with the first mode;

a first stack associated with the first mode;

a second stack associated with the second mode;

one or more pointers associated with the first stack; and

a trap handler emulating operations required by the at least one trapped instruction,

the trap handler comprising a plurality of the instructions executable in the second mode,

the processor executing the plurality of the instructions in the second mode to emulate the at least one trapped instruction,

wherein the plurality of the instructions in the second mode are adapted to utilize the first stack using the one or more pointers associated with the first stack.

67. The system according to Claim 66, further comprising:

an instruction register associated uniquely with the first mode, the instruction register for holding the address of an instruction executed in the first mode;

wherein the plurality of the instructions in the second mode are adapted to utilize the instruction register.

68. The system according to Claim 66, wherein the one or more pointers comprise a member selected from the group comprising:

a stack pointer for pointing to the top of the first stack, and

a local variable pointer pointing into the first stack.

69. The system according to Claim 68, further comprising:

one or more dedicated registers, each dedicated register adapted to hold one of the one or more pointers.

70. The system according to Claim 69, wherein the first mode comprises a Java mode and the second mode comprises a platform dependant mode.

71. The system according to Claim 70, wherein the instructions executed in the Java mode comprise a JVM instruction set.

72. The system according to Claim 71, wherein the instructions executed in the Java mode additionally comprise one or more quick instructions.

73. The system according to Claim 72, wherein the at least one trapped instruction comprises an instruction to be quickened, and wherein at least one of the plurality of

instructions adapted to access the instruction register are adapted to modify the opcode and/or operand(s) of the instruction to be quickened.

74. A platform with which instructions are processed, comprising:
processing means for processing the instructions in a first mode and a second mode.

75. The platform in claim 74, wherein the instructions executed in a first mode comprise software instructions executed in a platform independent mode, and wherein the instructions executed in the second mode comprise software instructions executed in a platform dependant mode.

76. The platform in claim 74, wherein the instructions executed in a first mode comprise software instructions compatible with a JVM compatible instruction set.

77. The platform of claim 76, wherein the platform dependant mode provides JVM compatible instruction set emulation for the software instructions executed in the platform dependant mode.

78. The platform of claim 77, wherein the instructions executed in the platform dependant mode comprise instructions compiled to a native instruction set of the platform.

79. The platform of claim 78, wherein the second mode comprises a C language mode.

80. A method for processing instructions on a platform in a first mode and a second mode comprising the steps of:
providing a mode indicator to indicate the first mode and the second mode;
executing the instructions in platform independent mode when the mode indicator indicates the first mode; and
executing the instructions in a platform dependent mode when the mode indicator indicates the second mode.

81. The method in claim 80, wherein the instructions executed in a first mode comprise instructions compatible with a JVM compatible instruction set.

82. The method of claim 81, wherein the platform dependant mode provides JVM compatible instruction set emulation for the software instructions executed in the platform dependant mode.

83. The method of claim 82, wherein the instructions executed in the platform dependant mode comprise instructions compiled to a native instruction set of the platform.

84. The method of claim 83, wherein the second mode comprises a C language mode.

85. A multi-mode system for executing instructions, comprising:

a respective stack for each mode; and

a respective circuit adapted to process instructions in the respective mode using the respective stack,

wherein at least one of the respective circuits uses the stack of another mode.

86. The system according to Claim 19 or 20, wherein instructions of the first mode are restricted to a first memory address range, and instructions of the second mode are restricted to a second address range.

87. The system according to Claim 86, further comprising a program counter register adapted to hold the address of an instruction being executed, wherein the mode indicator is set by one or more bits of the address in the program register such that when the program counter register contains an address in the first memory address range the mode indicator holds a first state; and when the program counter register contains an address in the second memory address range the mode indicator holds a second state.

88. The system according to Claim 87, wherein one or more bits of the program counter register are coupled to the mode indicator.

89. The system according to Claim 87, wherein

the state of the mode indicator changes from the first state to the second state when:

the execution of a branch instruction, stored in the first memory address range, adapted to branch execution to an address in the second address range, and

the state of the mode indicator changes from the second state to the first state when:

the execution of a branch instruction, stored in the second memory address range, adapted to branch execution to an address in the first memory address range.

Figure 1

Prior Art

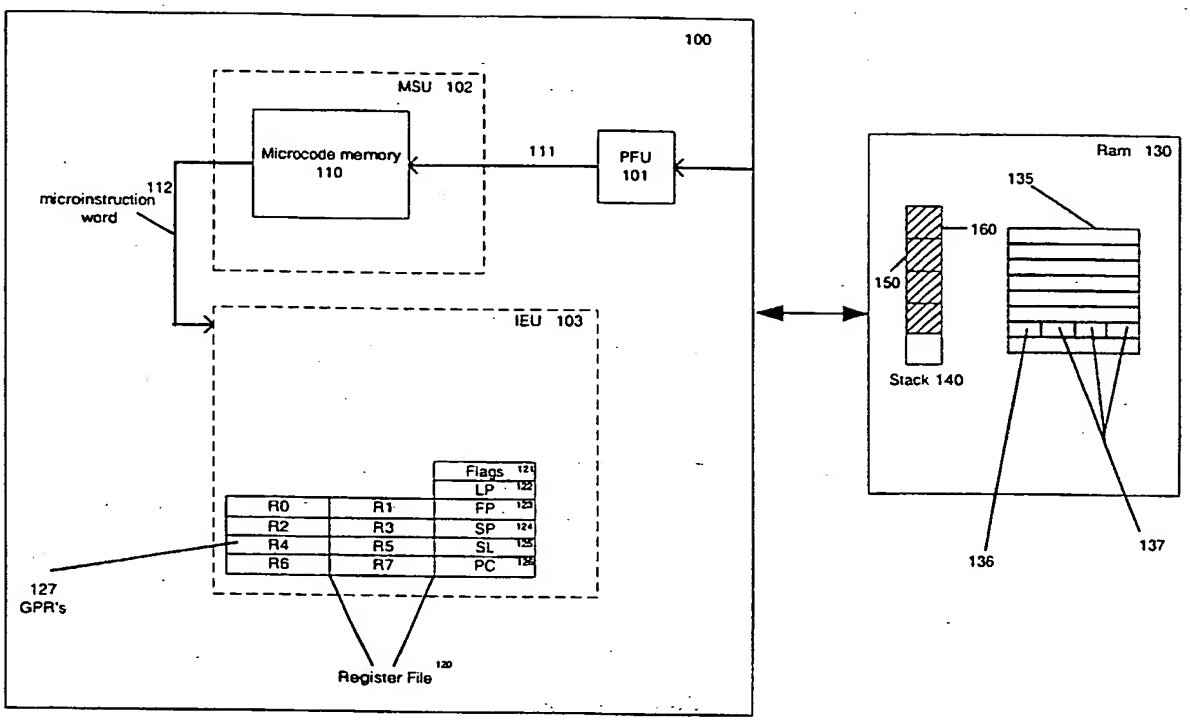


Figure 2A

Prior Art

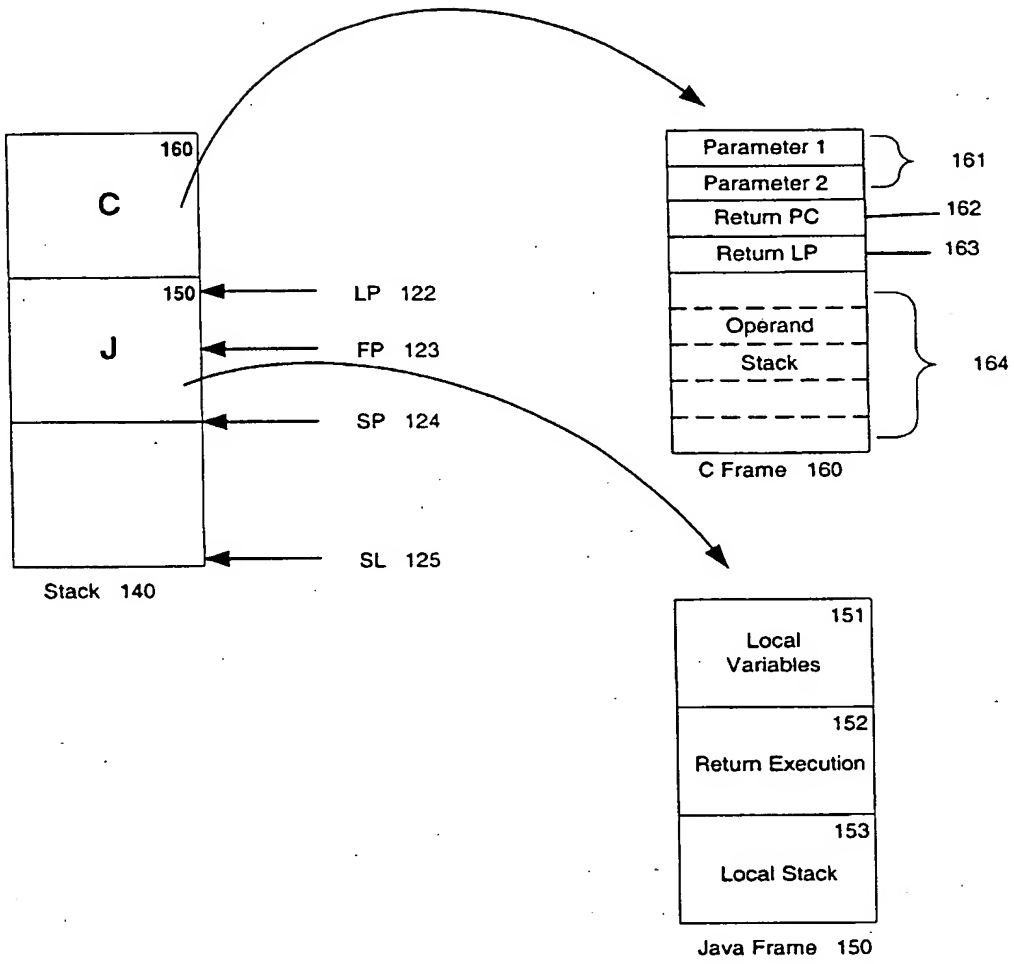


Figure 2B

Prior Art

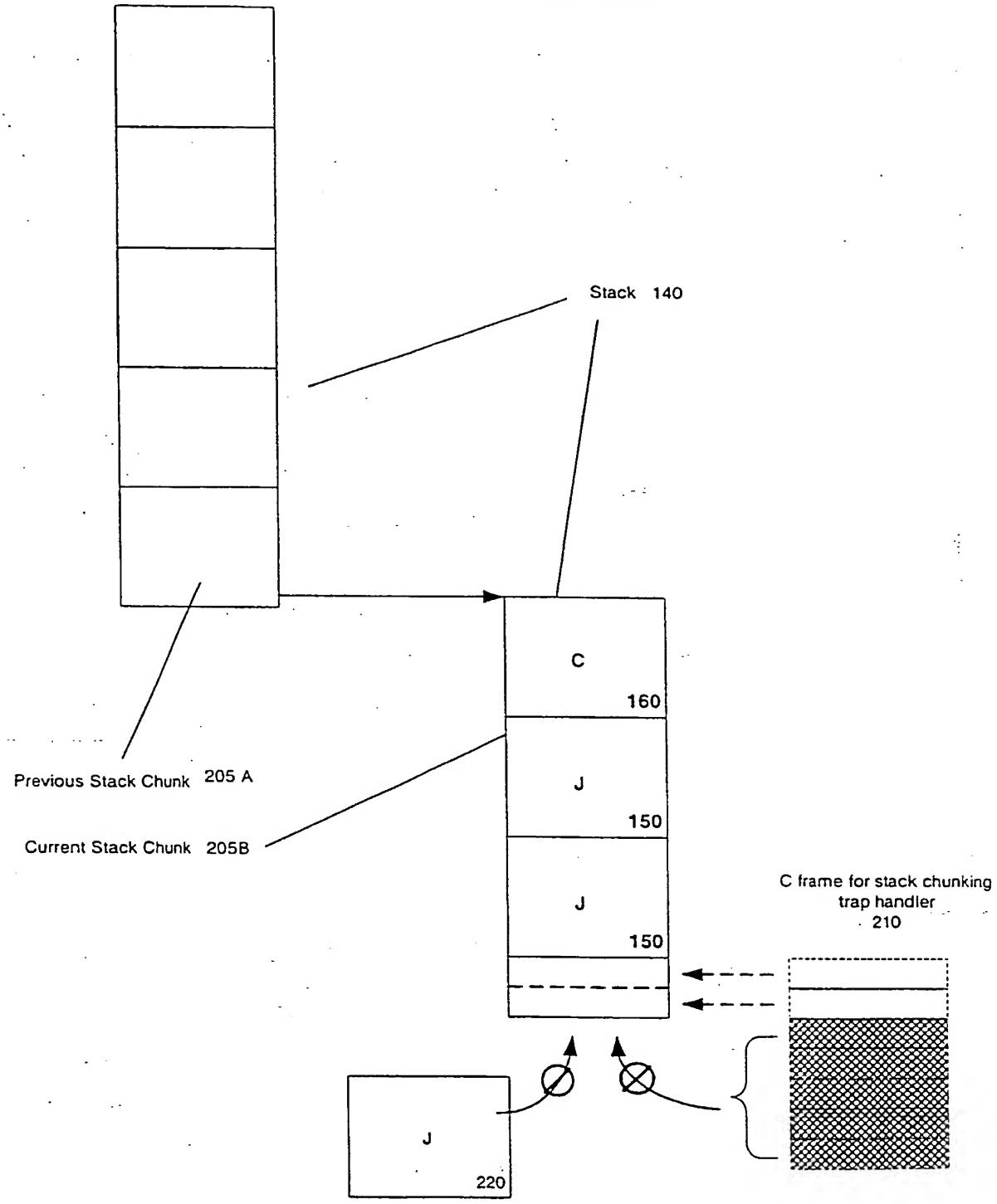


Figure 2C

Prior Art

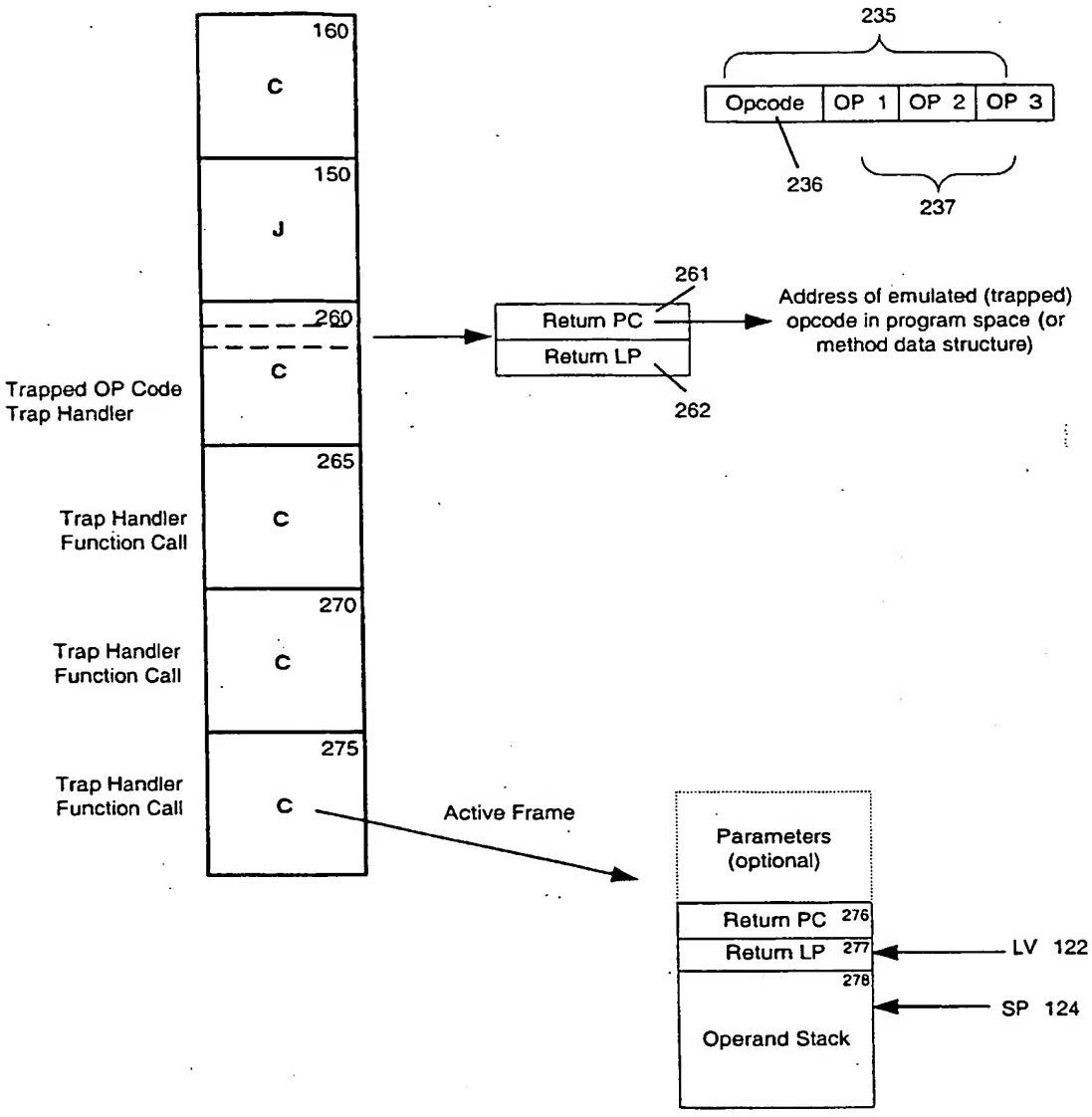


Figure 3

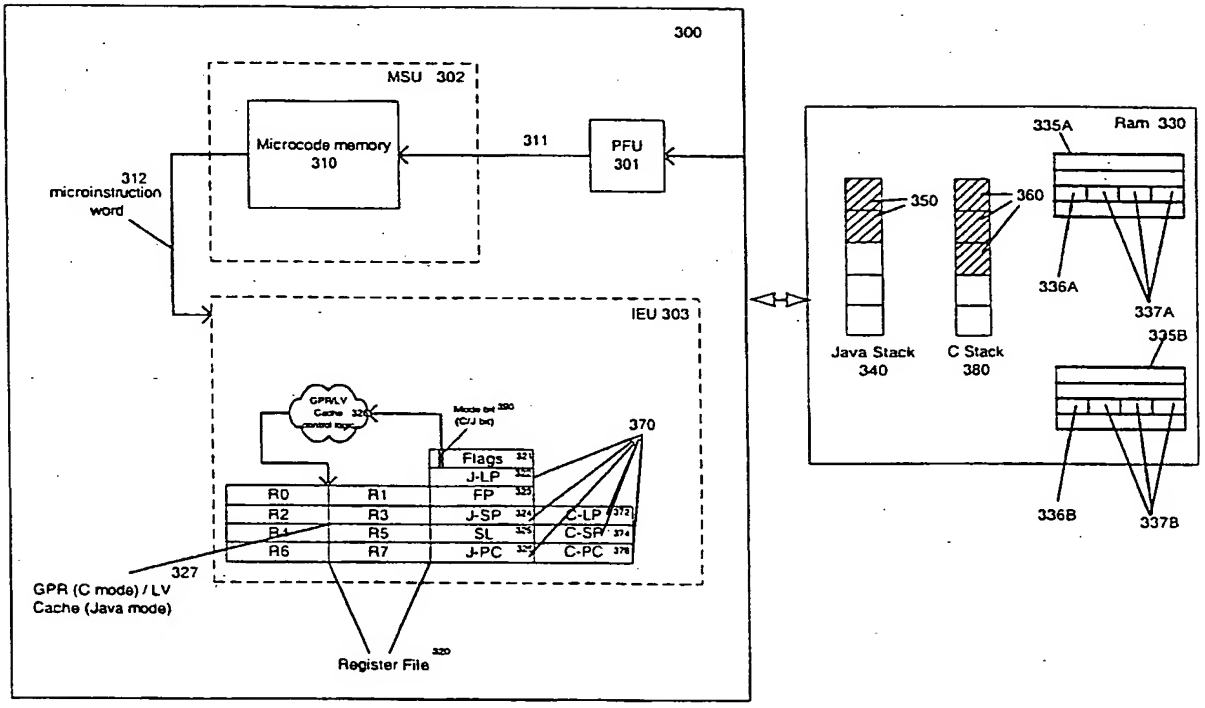


Figure 4A

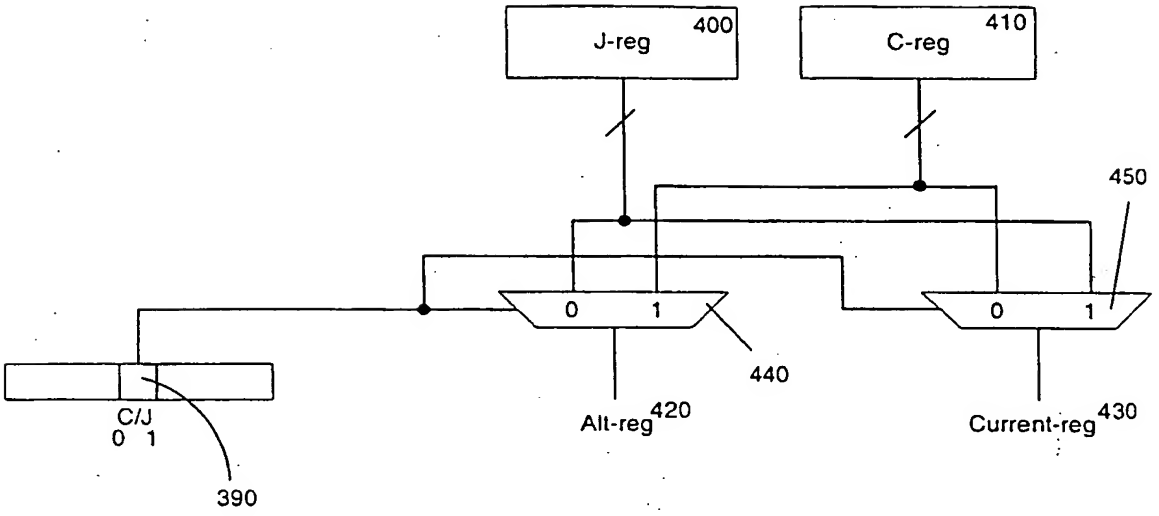


Figure 4B

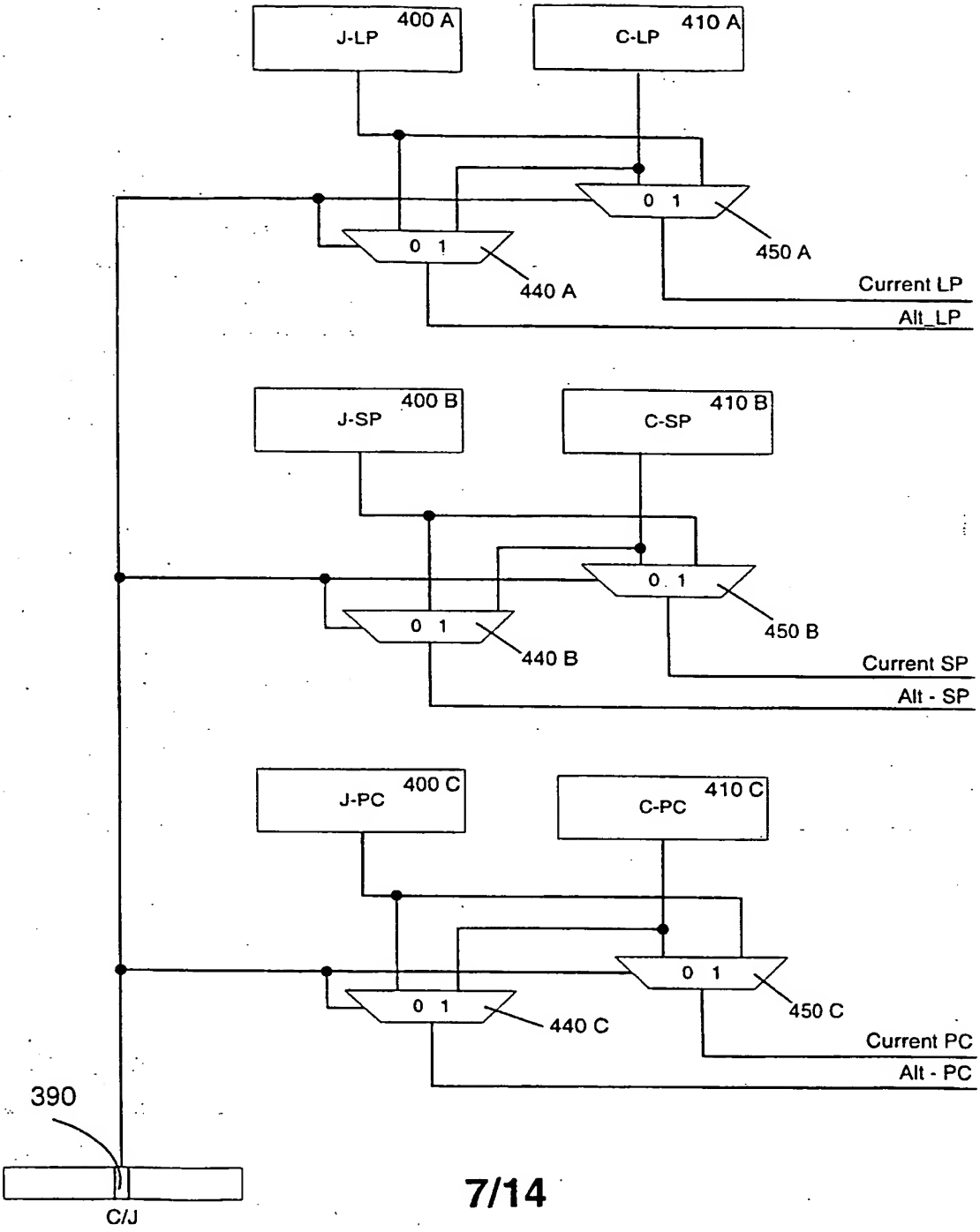


Figure 5

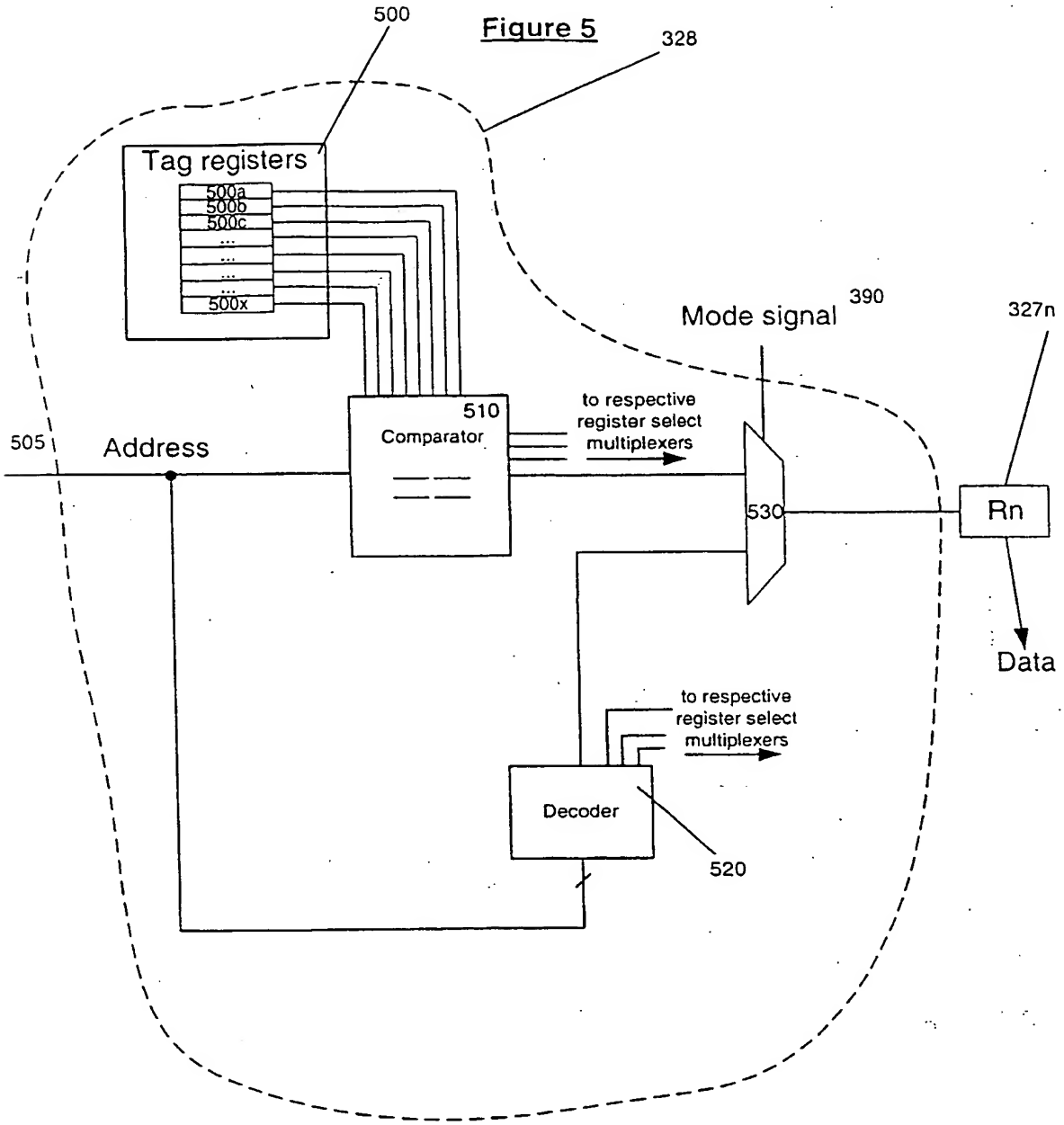


Figure 6A

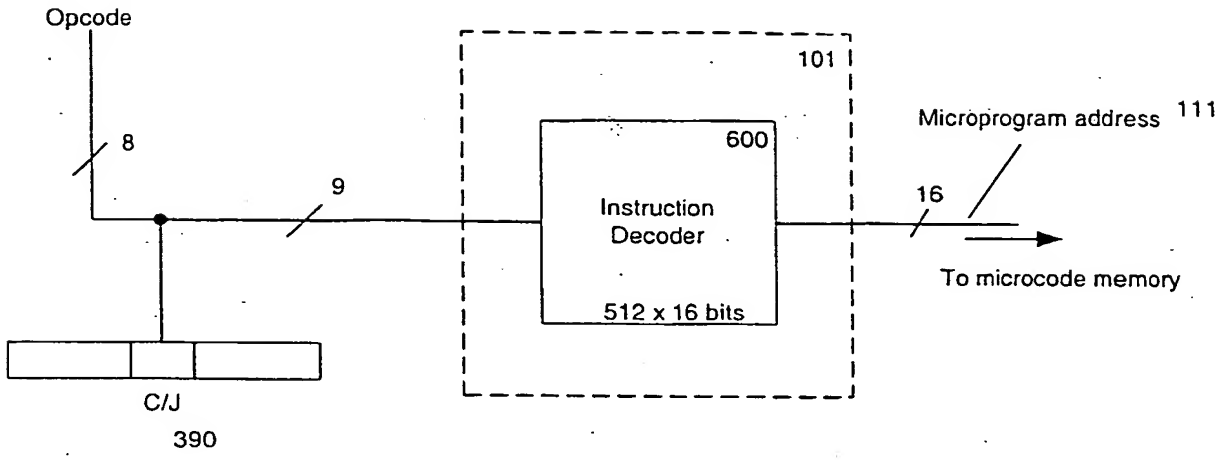


Figure 6B

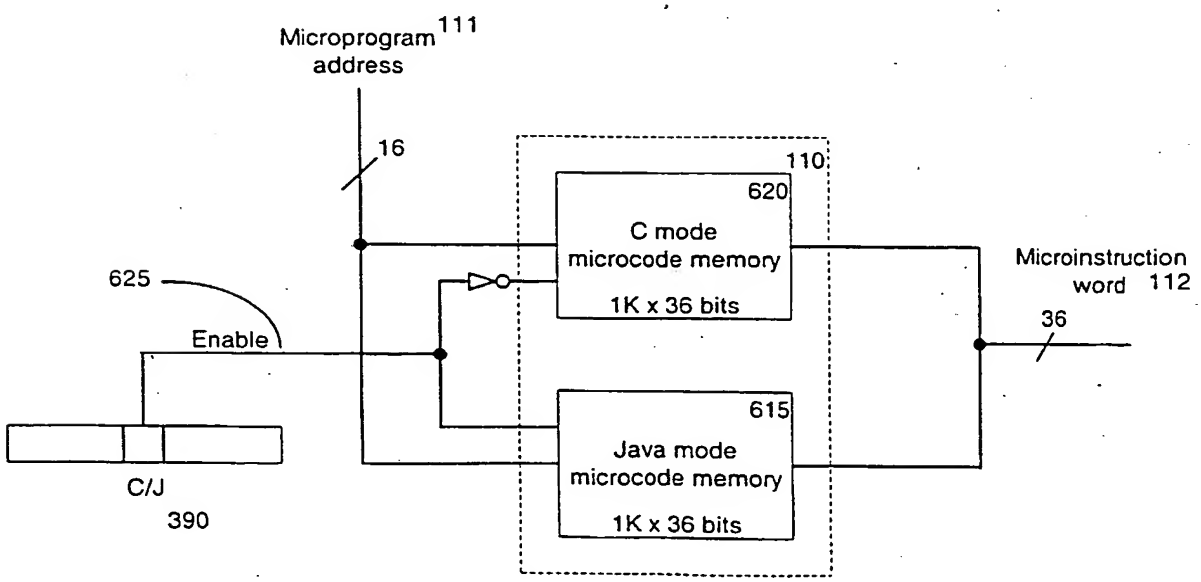


Figure 7

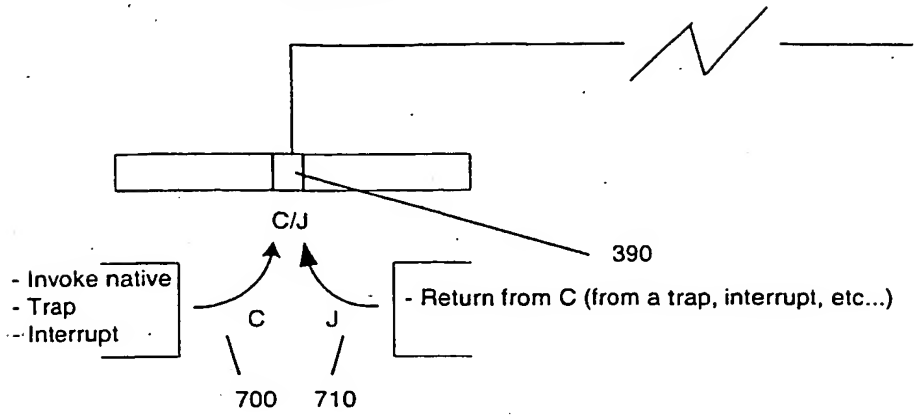
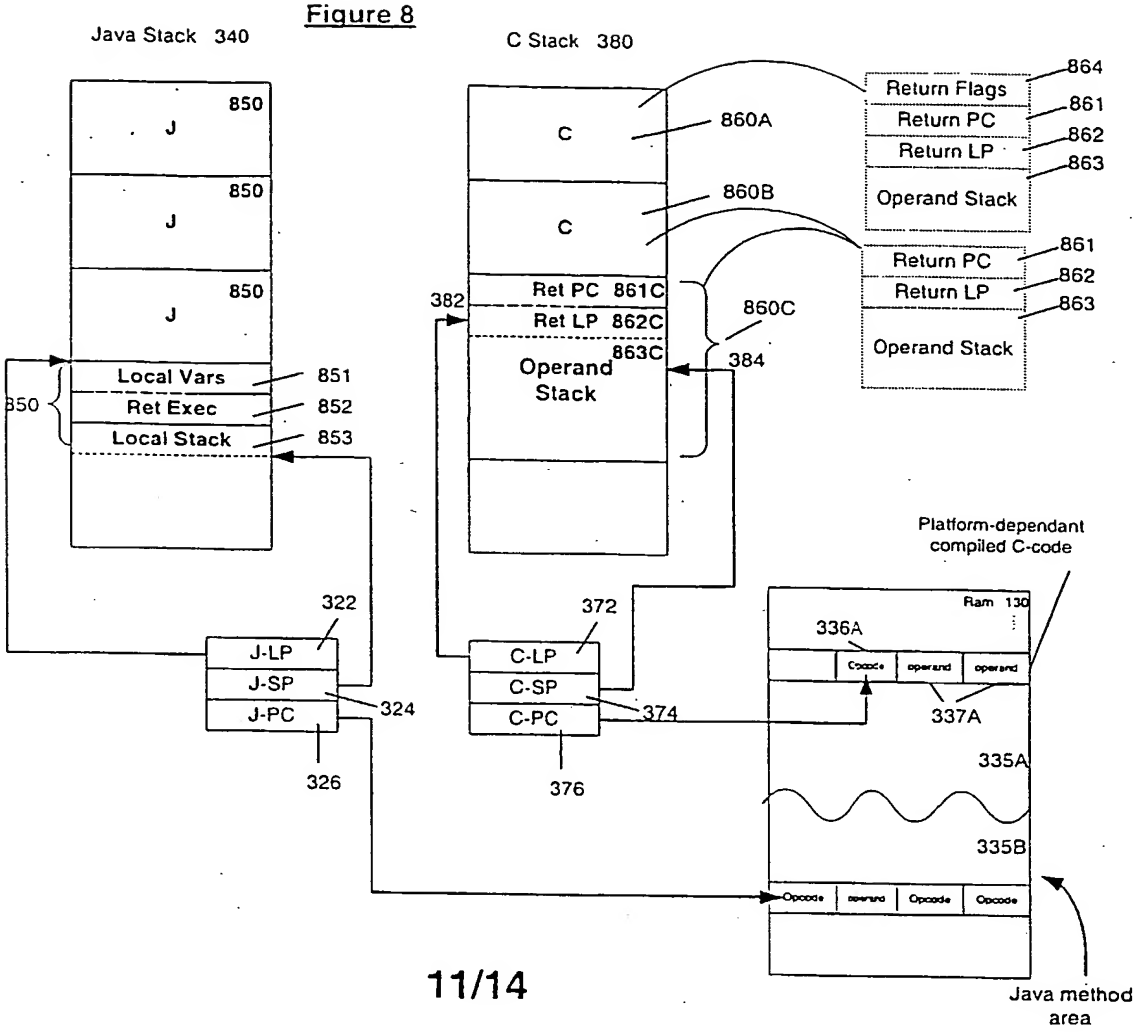


Figure 8



11/14

Figure 9

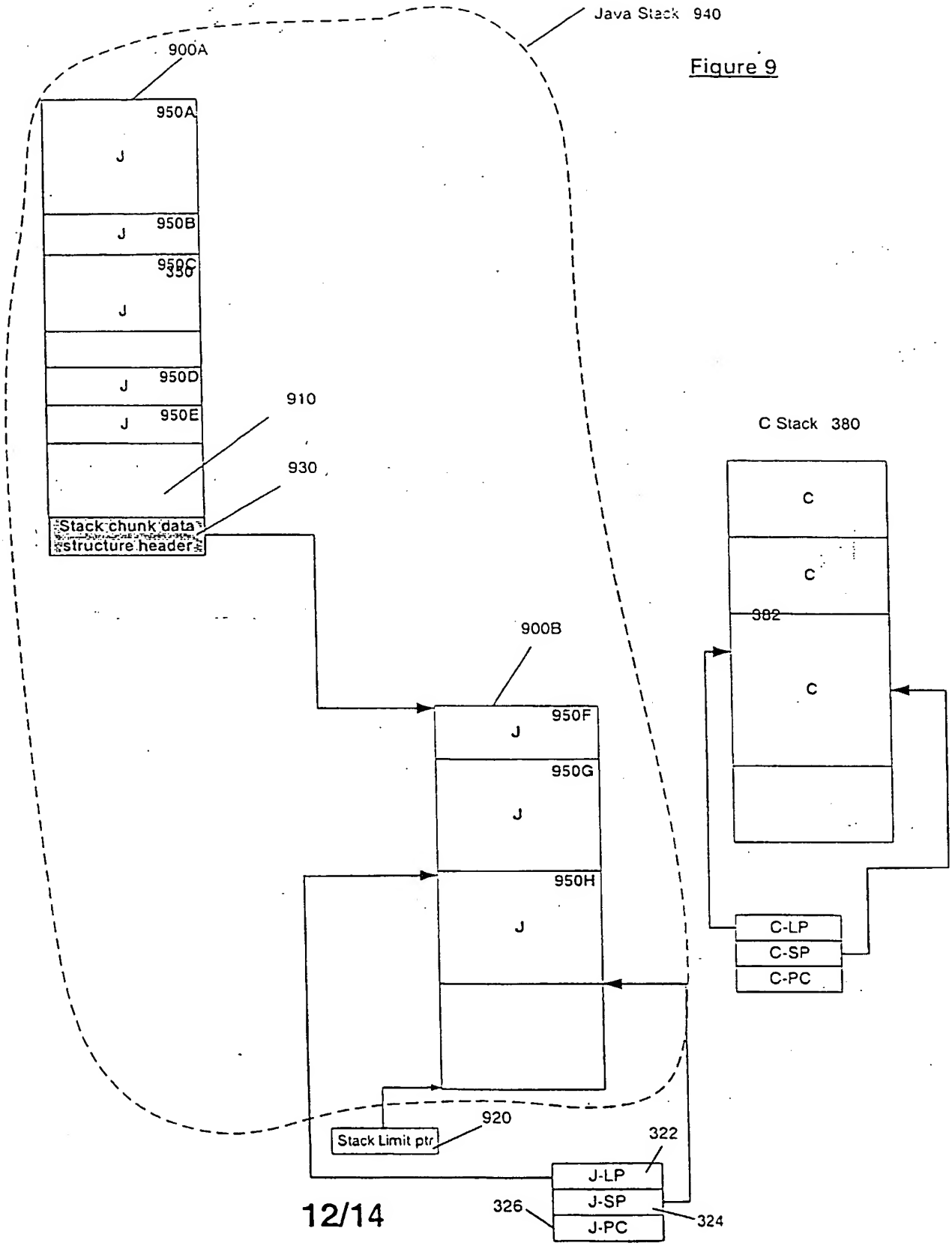


Figure 10

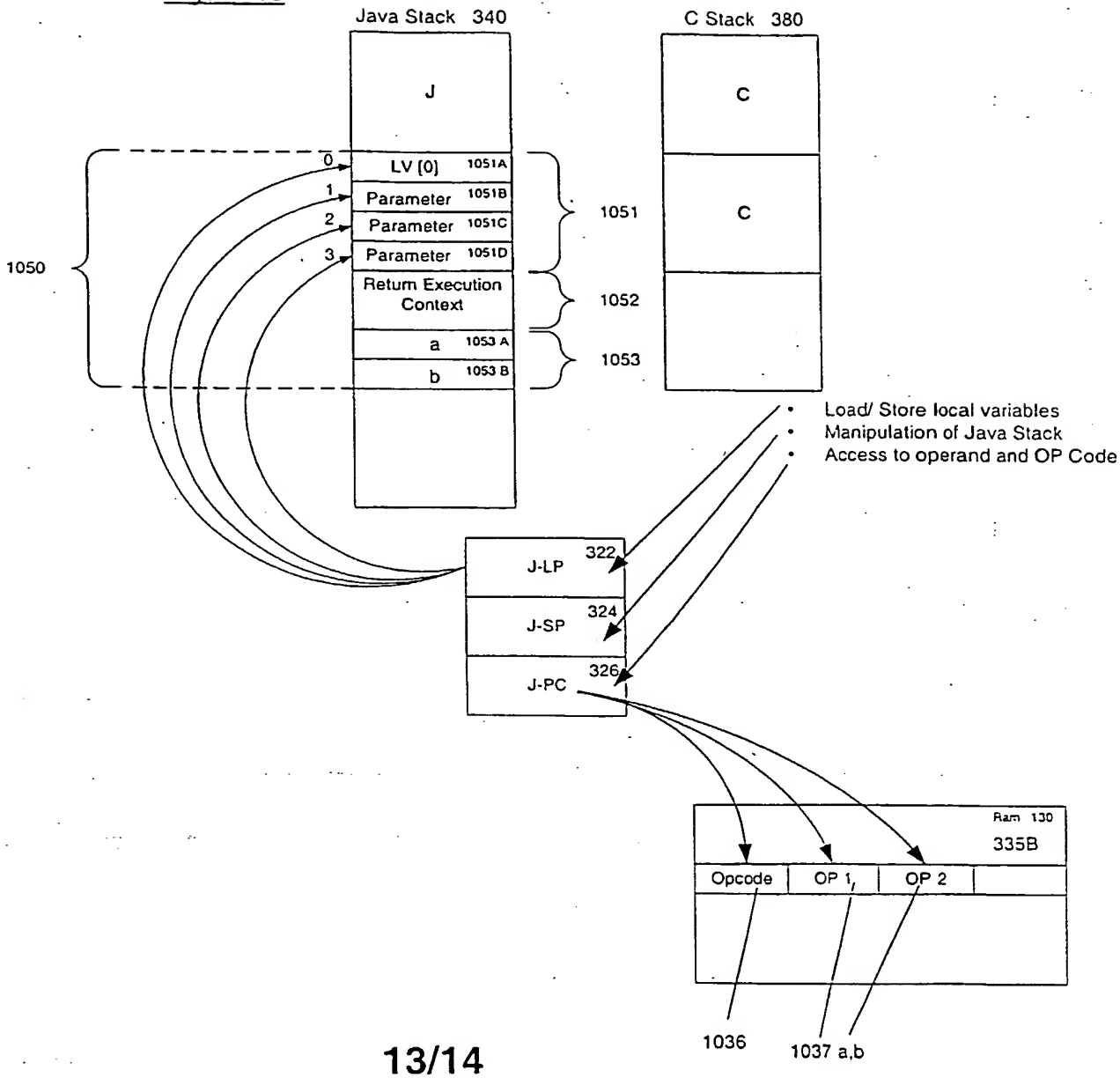
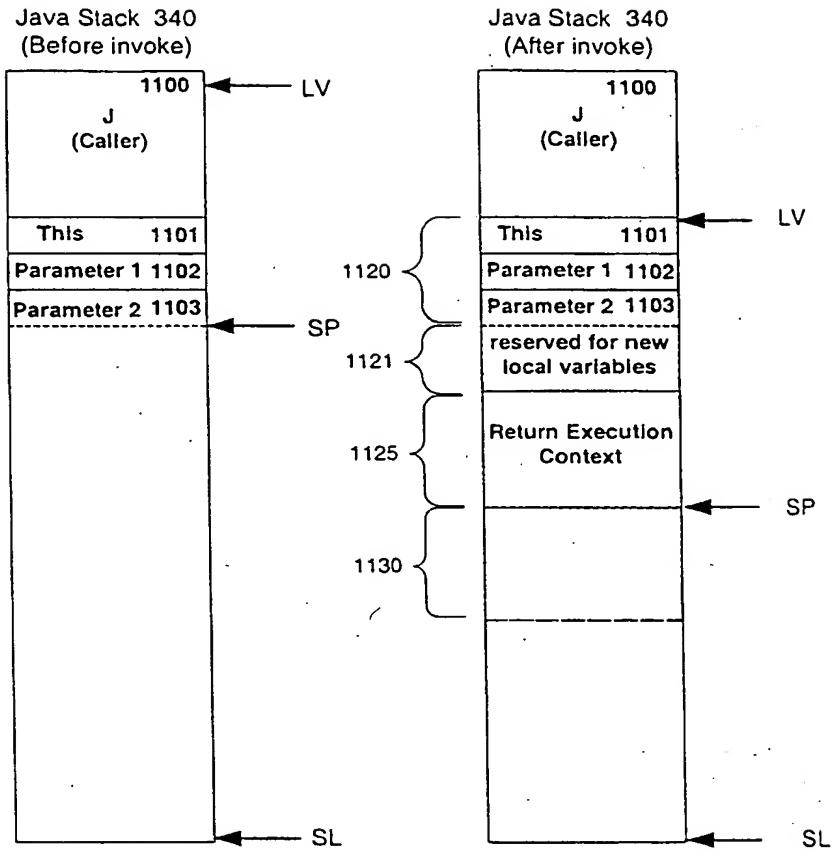


Figure 11



(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
12 September 2002 (12.09.2002)

PCT

(10) International Publication Number
WO 02/071211 A3

(51) International Patent Classification[7]: G06F 9/40, 9/455, 9/46

(21) International Application Number: PCT/US01/44031

(22) International Filing Date:
20 November 2001 (20.11.2001)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:

60/252,170	20 November 2000 (20.11.2000)	US
60/256,550	18 December 2000 (18.12.2000)	US
60/270,696	22 February 2001 (22.02.2001)	US
60/276,375	16 March 2001 (16.03.2001)	US
60/290,520	11 May 2001 (11.05.2001)	US
60/323,022	14 September 2001 (14.09.2001)	US
PCT/US01/43829	20 November 2001 (20.11.2001)	US
PCT/US01/43444	20 November 2001 (20.11.2001)	US
PCT/US01/44035	20 November 2001 (20.11.2001)	US
PCT/US01/43957	20 November 2001 (20.11.2001)	US

(71) Applicant: **ZUCOTTO WIRELESS, INC.** [US/US]; 4225 Executive Square, Suite 1040, La Jolla, CA 92037 (US).

(72) Inventors; and

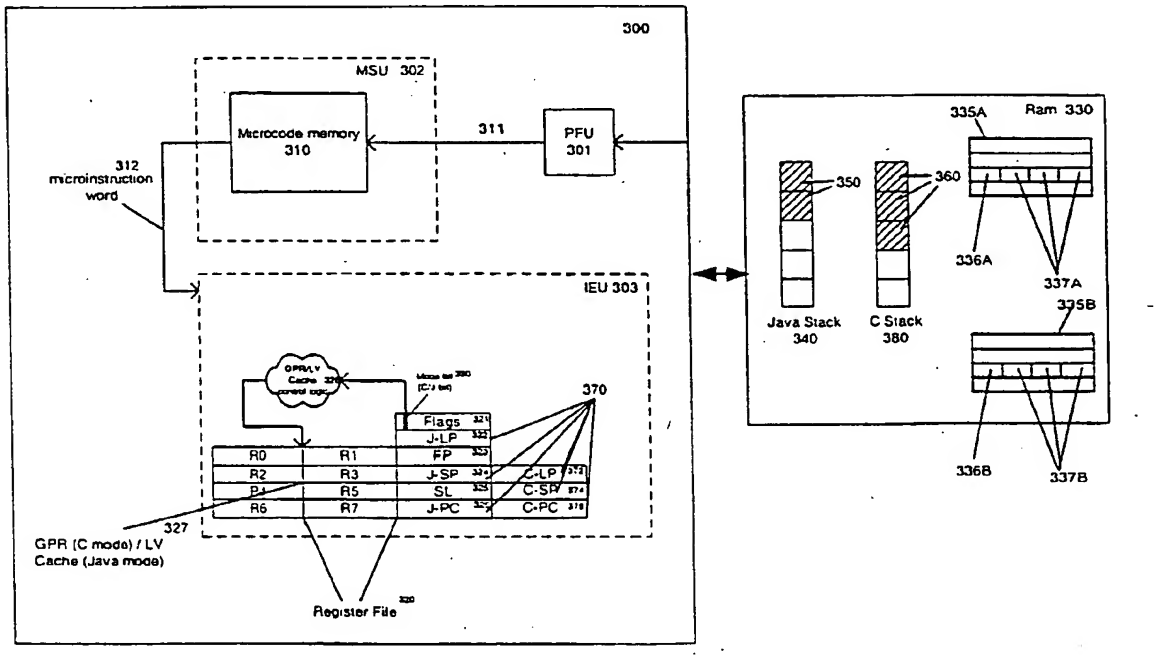
(75) Inventors/Applicants *(for US only)*: **MAJID, Michael** [CA/CA]; 106 Halley Str., Nepean, Ontario K2J3R9 (CA). **SAHRAOUI, Zohair** [CA/CA]; 6243 Castille Court, Gloucester, Ontario K1C1X4 (CA). **COUTURE, Mark** [CA/CA]; 13 Mystic Pvt., Ottawa, Ontario K1V1K5 (CA). **ROUFFER, Christopher** [CA/CA]; 133 Hamilton Ave., Ottawa, Ontario K1Y1C1 (CA).

(74) Agent: **WARDAS, Mark**; 4225 Executive Square, Suite 1040, La Jolla, CA 92037 (US).

(81) Designated States *(national)*: AE, AG, AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, BZ, CA, CH, CN, CO, CR, CU,

[Continued on next page]

(54) Title: DATA PROCESSOR HAVING MULTIPLE OPERATING MODES



(57) Abstract: A system executes instructions in two modes using two stacks. In the first mode a first stack is used, as well as an instruction accelerator for JVM instruction set. In the second mode both stacks are used, and the instructions are platform dependant and provide JVM instruction emulation. The first stack consists of chunks for threads, the second stack is statically allocated.

WO 02/071211 A3

CZ, DE, DK, DM, DZ, EC, EE, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MA, MD, MG, MK, MN, MW, MX, MZ, NO, NZ, PH, PL, PT, RO, RU, SD, SE, SG, SI, SK, SL, TJ, TM, TR, TT, TZ, UA, UG, US, UZ, VN, YU, ZA, ZW.

(84) **Designated States** *(regional)*: ARIPO patent (GH, GM, KE, LS, MW, MZ, SD, SL, SZ, TZ, UG, ZM, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE, TR), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, ML, MR, NE, SN, TD, TG).

Declaration under Rule 4.17:

— *of inventorship (Rule 4.17(iv)) for US only*

Published:

— *with international search report*

— *before the expiration of the time limit for amending the claims and to be republished in the event of receipt of amendments*

(88) **Date of publication of the international search report:**
25 September 2003

For two-letter codes and other abbreviations, refer to the "Guidance Notes on Codes and Abbreviations" appearing at the beginning of each regular issue of the PCT Gazette.

INTERNATIONAL SEARCH REPORT

Inte ial Application No
PCT/US 01/44031

A. CLASSIFICATION OF SUBJECT MATTER
IPC 7 G06F9/40 G06F9/455 G06F9/46

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
IPC 7 G06F

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practical, search terms used)
EPO-Internal, INSPEC, COMPENDEX

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category °	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
X	PETER STUHLMÜLLER: "16Bit Generation Z8000 Aufbau und Anwendung" 1980 , TE-WI VERLAG GMBH , MÜNCHEN XP002238068 ISBN: 3-921803-07-1 page 2-53, left-hand column, line 49 - line 51	1-8,20, 74-86
A		9-31, 87-89
Y	page 3-3; figure 3.1 page 5-22, right-hand column, line 30 - line 51 page 5-27; figure 5.16 page 5-2, right-hand column, line 13 - line 19 page 5-9, left-hand column, line 43 -right-hand column, line 21 --- -/--	66-73

[X] Further documents are listed in the continuation of box C.

[X] Patent family members are listed in annex.

° Special categories of cited documents :

"A" document defining the general state of the art which is not considered to be of particular relevance

"E" earlier document but published on or after the international filing date

"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)

"O" document referring to an oral disclosure, use, exhibition or other means

"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art.

"&" document member of the same patent family

Date of the actual completion of the international search	Date of mailing of the international search report
14 April 2003	22.07.03

Name and mailing address of the ISA	Authorized officer
European Patent Office, P.B. 5818 Patentlaan 2 NL - 2280 HV Rijswijk Tel. (+31-70) 340-2040, Tx. 31 651 epo nl, Fax: (+31-70) 340-3016	Müller, T

Form PCT/ISA/210 (second sheet) (July 1992)

C.(Continuation) DOCUMENTS CONSIDERED TO BE RELEVANT

Category *	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
Y	SHIMIZU N ET AL: "A dual issue queued pipelined Java processor TRAJA-toward an open source processor project" ASICS, 1999. AP-ASIC '99. THE FIRST IEEE ASIA PACIFIC CONFERENCE ON SEOUL, SOUTH KOREA 23-25 AUG. 1999, PISCATAWAY, NJ, USA,IEEE, US, 23 August 1999 (1999-08-23), pages 213-216, XP010371870 ISBN: 0-7803-5705-1 page 213, right-hand column, line 17 - line 21 ---	66-73
P,X	VIJAYKRISHNAN N ET AL: "Supporting object accesses in a Java processor" IEE PROCEEDINGS E. COMPUTERS & DIGITAL TECHNIQUES, INSTITUTION OF ELECTRICAL ENGINEERS. STEVENAGE, GB, vol. 147, no. 6, 28 November 2000 (2000-11-28), pages 435-443, XP006013937 ISSN: 1350-2387 the whole document ---	15,31
X	EP 0 840 210 A (SUN MICROSYSTEMS INC) 6 May 1998 (1998-05-06)	9
A	the whole document ---	10-31
X	EP 0 622 732 A (TOSHIBA MICRO ELECTRONICS ;TOKYO SHIBAURA ELECTRIC CO (JP)) 2 November 1994 (1994-11-02) abstract; claim 1; figure 1 ---	33-36
A	KIMURA S ET AL: "An application specific Java processor with reconfigurabilities" DESIGN AUTOMATION CONFERENCE, 2000. PROCEEDINGS OF THE ASP-DAC 2000. ASIA AND SOUTH PACIFIC YOKOHAMA, JAPAN 25-28 JAN. 2000, PISCATAWAY, NJ, USA,IEEE, US, 25 January 2000 (2000-01-25), pages 25-26, XP010376307 ISBN: 0-7803-5973-9 page 25, right-hand column, line 9 - line 12 --- -/--	66-73

INTERNATIONAL SEARCH REPORT

Inter al Application No
PCT/US 01/44031

C.(Continuation) DOCUMENTS CONSIDERED TO BE RELEVANT

Category *	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
A	EL-KHARASHI M W ET AL: "Java microprocessors: computer architecture implications" COMMUNICATIONS, COMPUTERS AND SIGNAL PROCESSING, 1997. 10 YEARS PACRIM 1987-1997 - NETWORKING THE PACIFIC RIM. 1997 IEEE PACIFIC RIM CONFERENCE ON VICTORIA, BC, CANADA 20-22 AUG. 1997, NEW YORK, NY, USA,IEEE, US, 20 August 1997 (1997-08-20), pages 277-280, XP010244969 ISBN: 0-7803-3905-3 the whole document ---	1-31, 33-36, 66-89
A	BRINKSCHULTE U ET AL: "A multithreaded Java microcontroller for thread-oriented real-time event-handling" PARALLEL ARCHITECTURES AND COMPILATION TECHNIQUES, 1999. PROCEEDINGS. 1999 INTERNATIONAL CONFERENCE ON NEWPORT BEACH, CA, USA 12-16 OCT. 1999, LOS ALAMITOS, CA, USA,IEEE COMPUT. SOC, US, 12 October 1999 (1999-10-12), pages 34-39, XP010360150 ISBN: 0-7695-0425-6 the whole document ---	1-31, 33-36, 66-89
A	MCGHAN H ET AL: "PICOJAVA: A DIRECT EXECUTION ENGINE FOR JAVA-BYTECODE" COMPUTER, IEEE COMPUTER SOCIETY, LONG BEACH., CA, US, US, vol. 31, no. 10, October 1998 (1998-10), pages 22-30, XP000859744 ISSN: 0018-9162 the whole document -----	1-31, 33-36, 66-89

INTERNATIONAL SEARCH REPORT

International application No.
PCT/US 01/44031

Box I Observations where certain claims were found unsearchable (Continuation of item 1 of first sheet)

This International Search Report has not been established in respect of certain claims under Article 17(2)(a) for the following reasons:

1. ☐ Claims Nos.:
because they relate to subject matter not required to be searched by this Authority, namely:

2. ☐ Claims Nos.:
because they relate to parts of the International Application that do not comply with the prescribed requirements to such an extent that no meaningful International Search can be carried out, specifically:

3. ☐ Claims Nos.:
because they are dependent claims and are not drafted in accordance with the second and third sentences of Rule 6.4(a).

Box II Observations where unity of invention is lacking (Continuation of item 2 of first sheet)

This International Searching Authority found multiple inventions in this international application, as follows:

see additional sheet

1. ☐ As all required additional search fees were timely paid by the applicant, this International Search Report covers all searchable claims.

2. ☐ As all searchable claims could be searched without effort justifying an additional fee, this Authority did not invite payment of any additional fee.

3. ☐ As only some of the required additional search fees were timely paid by the applicant, this International Search Report covers only those claims for which fees were paid, specifically claims Nos.:

4. ☒ No required additional search fees were timely paid by the applicant. Consequently, this International Search Report is restricted to the invention first mentioned in the claims; it is covered by claims Nos.:

1-31, 33-36, 66-73, 74-89

Remark on Protest

☐ The additional search fees were accompanied by the applicant's protest.

☐ No protest accompanied the payment of additional search fees.

FURTHER INFORMATION CONTINUED FROM PCT/ISA/ 210

This International Searching Authority found multiple (groups of) inventions in this international application, as follows:

1. Claims: 1-31,33-36,74-89

   Using the stack of a first mode when executing instructions in a second mode additionally to the stack of the second mode in a system with two modes of instruction execution.

   1.1. Claims: 66-73
   Executing a platform-independent instruction of a first mode by trapping the instruction and emulating the instruction by executing a corresponding trap handler in a second mode.

2. Claim : 32

   Using a mode indicator as selector for values from a set of values corresponding to the respective mode in a system with multiple modes of instruction execution.

3. Claims: 37-41

   Using a set of registers in a first mode as cache and in a second mode as general purpose registers in a system with two modes of instruction execution.

4. Claims: 42-64

   An instruction decoder using mode bits in combination with fixed-length opcodes to increase the number of available opcodes.

5. Claim : 65

   Selecting the implemention of instructions depending on the current mode of execution indicated by a mode indicator in a system with multiple modes of instruction execution.

Please note that all inventions mentioned under item 1, although not necessarily linked by a common inventive concept, could be searched without effort justifying an additional fee.

INTERNATIONAL SEARCH REPORT

Information on patent family members

Inter ial Application No
PCT/US 01/44031

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
EP 0840210 A	06-05-1998	US 5835958 A	10-11-1998
		EP 0840210 A2	06-05-1998
		JP 10232785 A	02-09-1998
		SG 53110 A1	28-09-1998
EP 0622732 A	02-11-1994	JP 2883784 B2	19-04-1999
		JP 6309177 A	04-11-1994
		DE 69425086 D1	10-08-2000
		EP 0622732 A1	02-11-1994
		KR 138468 B1	15-06-1998
		US 5459682 A	17-10-1995